MEGA: Malleable Encryption Goes Awry

Matilda Backendal, Miro Haller and Kenneth G. Paterson

Department of Computer Science, ETH Zurich, Zurich, Switzerland

Email: {mbackendal, kenny.paterson}@inf.ethz.ch, miro.haller@alumni.ethz.ch

Abstract—**MEGA is a leading cloud storage platform with more than 250 million users and 1000 Petabytes of stored data. MEGA claims to offer user-controlled, end-to-end security. This is achieved by having all data encryption and decryption operations done on MEGA clients, under the control of keys that are only available to those clients. This is intended to protect MEGA users from attacks by MEGA itself, or by adversaries who have taken control of MEGA's infrastructure.**

We provide a detailed analysis of MEGA's use of cryptography in such a malicious server setting. We present five distinct attacks against MEGA, which together allow for a full compromise of the confidentiality of user files. Additionally, the integrity of user data is damaged to the extent that an attacker can insert malicious files of their choice which pass all authenticity checks of the client. We built proof-of-concept versions of all the attacks. Four of the five attacks are eminently practical. They have all been responsibly disclosed to MEGA and remediation is underway.

Taken together, our attacks highlight significant shortcomings in MEGA's cryptographic architecture. We present immediately deployable countermeasures, as well as longer-term recommendations. We also provide a broader discussion of the challenges of cryptographic deployment at massive scale under strong threat models.

I. INTRODUCTION

The cloud – for outsourcing of both computation and data storage – has become a very popular approach to address scaling and management problems in IT. This applies to both enterprise and consumer domains. In the latter case, the market offers a myriad of different cloud services, with products having different combinations of storage, computation and collaboration features, and making a range of security and privacy claims. The consumer storage market alone was valued at USD 13.6 billion in 2021.[1]

As a prominent example, MEGA[2] is a cloud storage and collaboration platform founded in 2013 offering "secure storage and communication" services. With over 250 million registered users, 10 million daily active users [1] and 1000 PB of stored data [2], MEGA is a significant player in the consumer domain. What sets them apart from their competitors such as DropBox, Google Drive, iCloud and Microsoft OneDrive is the claimed security guarantees: MEGA advertise themselves as "the privacy company" and promise *user-controlled end-to-end encryption* (UCE).

UCE refers to the fact that data uploaded to the MEGA cloud is encrypted, and that only the user who owns the data has access to the key (derived from the user's password) needed to decrypt. Thus, MEGA's main selling point is

confidentiality of user data even against MEGA themselves, as showcased in the following quote from their website [3]:

> "MEGA does not have access to your password or your data. Using a strong and unique password will ensure that your data is protected from being hacked and gives you total confidence that your information will remain just that – yours."

This implies a threat model in which the service provider itself should be considered potentially adversarial, and yet the service should remain secure. All the service is then trusted for is availability. This adversarial model provides an interesting setting for cryptanalysis: not only does the adversary have access to encrypted user keys and data, it can also interact with users through legitimate channels during steps like user authentication and file access.

This may seem a very strong adversarial model. However, we stress that it is consistent with the security claims made by MEGA themselves. Moreover, we must consider the possibility that even if MEGA is not adversarial, their systems may have been compromised by malicious third parties, for example nation state security agencies or hacking groups, who wish to gain access to users' data and files. Indeed, the sheer size of MEGA – and the likelihood of it attracting users who wish to protect highly sensitive data precisely because of the security the service claims to offer – surely make MEGA an attractive target. Additionally, UCE ensures that MEGA cannot be coerced into revealing user data, e.g. through subpoenas, since it is technically unable to do so.

In this work, we review the security of MEGA in this threat model and find significant issues in how it uses cryptography. These lead to devastating attacks on the confidentiality and integrity of user data in the MEGA cloud.

A. The MEGA Key Hierarchy

MEGA's approach to UCE begins with the user password, which acts as the root of the key hierarchy depicted in Figure 1. The MEGA client derives an authentication key and an encryption key from the password. The authentication key is used to identify users to MEGA. The encryption key encrypts a randomly generated master key, which in turn encrypts other key material of the user. Every account has a set of asymmetric keys: an RSA key pair for sharing data, a `Curve25519` key pair for exchanging chat keys for MEGA's chat functionality, and an `Ed25519` key pair for signing the other keys. Furthermore, the client generates a new key for every file or folder (collectively referred to as *nodes*) uploaded by the user. All keys are encrypted by the client with the
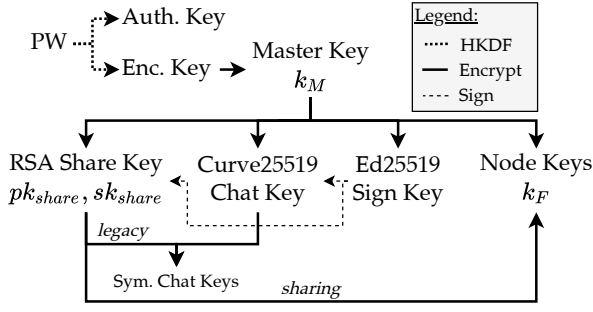

Fig. 1. MEGA's key hierarchy. The master key encrypts the share, chat, sign and node keys using AES-ECB.

master key using AES-ECB and then stored on MEGA's servers to support access from multiple devices. A user on a new device can enter their password, authenticate to MEGA, fetch the encrypted key material, and decrypt it with the encryption key derived from the password.

B. Trivial Attacks

The above description of MEGA's key hierarchy immediately leads to a trivial attack. MEGA could mount dictionary attacks on user passwords using data revealed in the authentication protocol (in which the authentication key is sent to the MEGA server). This attack is mitigated by users choosing strong passwords and by MEGA imposing a suitable password policy.

Moreover, another trivial attack is that MEGA could introduce malicious code to their web clients. This could be used to exfiltrate the user password or keys to the MEGA servers. MEGA provides a browser extension – including signed updates – which avoids loading code dynamically and instead runs it locally. This, to some extent, addresses this *code integrity* issue.

We do not consider these attacks any further in this paper, since they are both mitigated in the MEGA service.

C. Contributions

We present a series of five attacks on the key hierarchy of MEGA. The first two attacks exploit the lack of integrity protection of ciphertexts containing keys (henceforth referred to as *key ciphertexts*), and allow full compromise of all user keys encrypted with the master key, leading to a complete break of data confidentiality in the MEGA system. The next two attacks breach the integrity of file ciphertexts and allow a malicious service provider to insert chosen files into users' cloud storage. The last attack is a Bleichenbacher-style attack against MEGA's RSA encryption mechanism. It is applicable in a slightly weaker attack model than our first four attacks. We have developed proof-of-concept (PoC) implementations for all five attacks. We briefly present each attack next.

1) **RSA Key Recovery Attack.** Using the session ID exchange at the start of a client connection to MEGA, a malicious service provider can recover a user's private RSA share key (used to share file and folder keys) over

512 login attempts. When the RSA key has been compromised by the attacker, the confidentiality and integrity of all node keys shared with the victim is lost. Our attack exploits the lack of integrity protection of the encrypted RSA key and properties of the RSA-CRT implementation used by MEGA clients to build an oracle that leaks one bit of information per login attempt about a factor of the RSA modulus. It combines this novel attack vector with known lattice techniques to accelerate the attack.

2) **Plaintext Recovery Attack.** Building on the previous vulnerability, the malicious service provider can recover any plaintext encrypted with AES-ECB under a user's master key. This includes all node keys used for encrypting files and folders (including unshared ones not affected by the previous attack), as well as the private `Ed25519` signature and `Curve25519` chat key. As a consequence, the confidentiality of all user data protected by these keys, such as files and chat messages, is lost. This attack exploits MEGA's reuse of the master key and the use of RSA-CRT, in combination with the abilities of the adversary to manipulate key ciphertexts and choose plaintexts used in the authentication protocol. We believe it to be an entirely novel kind of attack.

3) **Two Integrity Attacks.** We present two attacks with which a malicious service provider can break the integrity of the file encryption scheme and insert arbitrary files into the user's file storage which pass the authenticity checks during decryption. This enables framing of the user by inserting controversial, illegal, or compromising material into their file storage. The attacks are non-trivial because the adversary cannot properly encrypt node keys without access to the user's master key.

The first attack uses the previous plaintext recovery attack to obtain a suitable node key and then constructs an encrypted file. The user cannot demonstrate that they did not upload the forged data because the files and keys are indistinguishable from genuinely uploaded ones.

The second integrity attack does not require that the attacker has access to a decryption oracle for AES-ECB under the master key. Instead, it exploits a fundamental problem with the method used by MEGA to "obfuscate" file and folder keys before encryption. It needs only knowledge of a single AES block and its AES-ECB encryption under the user's master key to create a forgery that is difficult to detect.

4) **RSA Decryption Attack.** The RSA encryption used to exchange chat keys as a legacy fallback is vulnerable to a novel variant of Bleichenbacher's attack on `PKCS#1 v1.5` padding [4]. This attack allows for the decryption of RSA ciphertexts containing chat and node keys. This is already implied by the key recovery attack in point 1, but this attack requires a weaker adversary model (which we describe in detail in the sequel). This attack is challenging to perform in practice as it would require almost 2^{17} client interactions. Nevertheless, it exposes an entirely independent attack vector and uncovers

additional issues in MEGA's cryptographic design.

In addition to these technical contributions, we propose countermeasures to protect against the attacks, as well as a discussion of more general learnings about the challenges of deploying and maintaining cryptography at scale.

D. Ethical Considerations

We contacted MEGA to inform them of the vulnerabilities in their system and to suggest three different levels of mitigation (immediate, minimal, and recommended) on March 24, 2022. We suggested a 90-day disclosure period. MEGA acknowledged the attacks on March 24, 2022, confirming that the system is vulnerable and needs patching. They decided to introduce additional client-side checks on the format of RSA private keys to protect against our first attack. While these checks directly prevent the RSA key recovery attack, and hence by extension the attacks that depend on it, this fix significantly differs from our proposed countermeasures. MEGA released their patches on June 21, 2022, and awarded us a bug bounty.

We only worked with our own test account when exploring MEGA's services and building our PoCs. We avoided overloading MEGA with login requests when running our attacks. We did not attempt to reverse engineer any MEGA server-side code, but instead relied on MEGA's whitepaper [5], inspection of client-side code, and the public server API.

E. Related Work

This paper is based on Haller's master thesis [6].

1) Previous Attacks on Cloud Storage Systems: Dalskov and Orlandi [7] performed an in-depth analysis of SpiderOak One, another provider with user-controlled encryption, in a threat model similar to ours. They uncovered several vulnerabilities in the cryptographic design of SpiderOak One which led to a breach of data confidentiality. Niehage [8] discovered four attacks on Nextcloud. The first vulnerability exploited that the server could maliciously replace public keys due to the lack of integrity protection. The other three attacks break file integrity by modifying files partially, replacing them, or performing a downgrade attack on the encryption.

2) Related Cryptographic Attacks: Previous results on key recovery through over-writing of key material targeted RSA in the context of OpenPGP [9]. The focus was on signatures instead of encryption with only partial output. A more systematic analysis of the impact of key over-writing attacks on OpenPGP was recently given in [10]. Power fault attacks on RSA signatures [11], [12] inspired our attack on RSA-CRT, however, we tamper with the private key instead of inducing errors in single computations. Prior work on authenticated encryption without key commitment constructs ciphertexts that can be decrypted to two valid files using different keys [13], [14], [15]. While they share the setting of AES primitives with known keys, we only consider a single key for our integrity attacks and target CBC-MAC instead of encryption.

We contribute an RSA decryption attack that is a novel variant of Bleichenbacher's attack on PKCS#1 v1.5 from 1998 [4]. Other instances of this attack [16], [17], [18], [19], [20] exploit different side-channel leakage to build padding oracles. Unlike them, we do not target PKCS#1 v1.5 padding but the custom padding scheme of MEGA that includes an unknown prefix circumventing the straightforward adaption of Bleichenbacher's attack.

3) Proposals for Cloud Storage Systems: A high-level survey of the functionality and security of multiple cloud storage providers is given in [21]. Messmer et al. [22] provide a generic security model for a simplified cloud file system. Boyd et al. [23] give models for secure cloud storage, including consideration of a compromised service provider. Kamara and Lauter [24] survey architectures for secure cloud storage with a trusted client and an untrusted service provider. Metal [25] and Titanium [26] are recent proposals for hiding metadata (as well as data) in encrypted file-sharing systems. These papers are part of a rich academic literature on secure cloud storage and collaboration systems.

F. Paper Organization

The next section gives a self-contained description of MEGA and its use of cryptography. The ensuing sections present our five attacks. Section VII describes our PoCs. Section VIII describes countermeasures to our attacks. Section IX discusses the wider implications of our work and future directions.

II. THE MEGA CLOUD

A. Notation

By $[m]_k$ we denote the encryption of a message m with the key k. The encryption algorithm can be derived from the context. We let $l[a{:}b]$ denote the slice $(e_a, e_{a+1}, \ldots, e_b)$ from the tuple $l \leftarrow (e_1, e_2, \ldots, e_n)$, where $1 \leq a \leq b \leq n$. We treat byte strings as tuples of bytes. We use $[a, b]$ to denote the integer set $\{a, a+1, \ldots, b\}$. B is shorthand for byte. By $\|$ and $\oplus$ we denote string concatenation and XOR, respectively.

B. Key Hierarchy

At the root of a MEGA client's key hierarchy, illustrated in Figure 1, is the password chosen by the user. From this and a client-chosen salt, two 128-bit keys are derived using `PBKDF2-HMAC-SHA512`: an *encryption key* k_e and an *authentication key* k_a. Additionally, the client generates a 128-bit master key k_M, which is encrypted with k_e using AES-ECB and uploaded to the server. Below the master key in the hierarchy reside the node keys:[3] a set of symmetric AES keys used to encrypt user files and folders (nodes). A fresh node key is generated for each node object created by the user. Moreover, each user has three asymmetric key pairs:

- "Share key": an RSA key pair for sharing node keys (and as a fallback solution for chat key transfer)
- "Chat key": a `Curve25519` key pair for exchanging keys for the MEGAchat

---

[3]Sometimes referred to as *data encryption keys* in other settings.

```
EncNode(k_M, F, attr):

    Given: master key k_M, node F, attributes attr
    Returns: encrypted file chunks [F_i]_{k_F}, attributes [attr]_{k_F},
    obfuscated key [k_F^{obf}]_{k_M}
 1  k_F ←$ {0,1}^128   // node key
 2  N_F ←$ {0,1}^64    // node nonce
 3  l_F ← 2^j for j ∈ [10, 13]   // #AES blocks per chunk
 4  F_1‖F_2‖ . . . ‖F_n ← F   // ∀i ∈ [1, n]. |F_i| = 128 · l_F bits
 5  For all i ∈ [1, n]:
 6     [F_i]_{k_F}, T_i ← AES-CCM*.Enc(k_F, N_F‖(i · l_F), F_i)
 7  C_F ← [F_1]_{k_F}‖[F_2]_{k_F}‖ . . . ‖[F_n]_{k_F}
 8  T_cond ← CBC-MAC.Tag(k_F, T_1‖T_2‖ . . . ‖T_n)
 9  k_F^{obf} ← ObfKey(k_F, N_F, T_cond)
10  [k_F^{obf}]_{k_M} ← AES-ECB.Enc(k_M, k_F^{obf})
11  [attr]_{k_F} ← AES-CBC.Enc(k_F, iv := 0^128, attr)
12  return C_F, [attr]_{k_F}, [k_F^{obf}]_{k_M}
```

Fig. 2. MEGA's chunkwise file encryption procedure.

```
AES-CCM*.Enc(k_F, IV, F_i):

    Given: node key k_F, file chunk F_i, initialization vector IV
    Returns: encrypted file chunk c_i, authentication tag T_i
 1  N_F ← IV[1:8]   // extract file nonce from leftmost 8B of IV
 2  T_i ← CBC-MAC.Tag(k_F, iv := N_F‖N_F, F_i)
 3  c_i ← AES-CTR.Enc(k_F, IV, F_i)
 4  return c_i, T_i
```

Fig. 3. MEGA's custom AES-CCM implementation.

- "Sign key": an Ed25519 key pair for signing the other public keys

The private keys and the node keys are encrypted with AES-ECB under the master key and the resulting ciphertexts are stored by MEGA's servers.

C. Node Encryption

To encrypt a file F, the client first samples a random 128-bit node key k_F and a 64-bit nonce N_F. Large files are then partitioned into chunks F_i of size between 128 KB and 1 MB.[4] Consequently, there are between 2^{10} and 2^{13} AES blocks per chunk, each 128 bits large. The chunks are encrypted with a custom implementation of AES-CCM with key k_F and nonce N_F. Additionally, the file attributes $attr$ (containing metadata such as the filename) are encrypted using AES-CBC with a zero IV and key k_F. The full node encryption procedure is shown in Figure 2. For folders, which do not have file content, the file input F is ignored and only the attributes are encrypted.

Figure 3 describes AES-CCM*, MEGA's variant of AES-CCM. This deviates from the specification in RFC 3610 [27],[5] which invalidates the formal security analysis in [29]. However, we did not find attacks on AES-CCM*.

---

[4]Clients usually select a single chunk size and use it for all chunks.

[5]The CBC-MAC tag of the plaintext chunk F_i is computed using the IV $N_F‖N_F$, rather than the zero bytes string (cf. RFC 3602 [28]). Furthermore, the CBC-MAC tag is returned in the clear, rather than encrypted with the first key stream block from the counter mode encryption. This means that MEGA's variant of AES-CCM is effectively an Encrypt-and-MAC scheme, rather than MAC-then-Encrypt.

```
ObfKey(k_F, N_F, T_cond):

    Given: node key k_F, file nonce N_F, condensed MAC T_cond
    Returns: obfuscated file key
 1  τ_1‖τ_2‖τ_3‖τ_4 ← T_cond // |τ_i| = 4, ∀i ∈ [0, 3]
 2  T_meta ← τ_1 ⊕ τ_2‖τ_3 ⊕ τ_4
 3  x ← k_F ⊕ (N_F‖T_meta)
 4  return x‖N_F‖T_meta
```

```
DeobfKey(k_F^{obf}):

    Given: obfuscated file key k_F^{obf}
    Returns: node key k_F, file nonce N_F, metamac T_meta
 5  x‖N_F‖T_meta ← k_F^{obf}
 6  k_F ← x ⊕ (N_F‖T_meta)
 7  return k_F, N_F, T_meta
```

Fig. 4. MEGA's key obfuscation and de-obfuscation procedures.

To finish the node encryption in Figure 2, the client aggregates the file chunk MACs T_1, T_2, …, T_n into a single condensed tag value T_{cond} using CBC-MAC with the key k_F applied to the concatenation of all chunk MACs. Additionally, the client computes an "obfuscated file key", k_F^{obf}, from the node key k_F, nonce N_F, and condensed tag T_{cond}. This k_F^{obf} is then encrypted with AES-ECB under the master key and uploaded to MEGA's servers together with the encrypted attributes and file chunks. Note that no MAC tag is computed for the attributes, implying a lack of integrity protection.

D. Key Obfuscation

MEGA applies an obfuscation procedure to the node key, nonce and MAC tag before they are encrypted and uploaded to the server. The obfuscation, described in Figure 4, aggregates the condensed MAC T_{cond} to a so-called "metamac" T_{meta} by splitting T_{cond} into four 4-byte chunks and XORing together the first two chunks, as well as the last two chunks. The key k_F is then XORed with the concatenation of N_F and T_{meta}. The final obfuscated key k_F^{obf} is obtained by appending N_F and T_{meta} to the scrambled file key.

Unfortunately, MEGA provides no reasoning for the design of the key obfuscation. We hypothesize that the aim is to create a binding between the involved components. However, as we show in Section V, the structure introduced by ObfKey leads to attacks on the integrity of node ciphertexts.

E. Authentication and Session ID Exchange

When a user logs into their MEGA account, the client derives the authentication key k_a from the password and sends it over a TLS connection to the server for authentication. The server compares the first 128 bits of the SHA-256 hash of k_a to a value stored during registration, indexed by the user's email address. If these values match, the server generates a session ID (SID) sid and pads it with two bytes to the left and 211 bytes to the right.[6] Let (pk_{share}, sk_{share}) be the 2048-bit RSA key pair of the user and let m be the padded sid.

---

[6]The exact padding scheme is not published by MEGA. However, the need for compatibility with the client-side decryption determines the position of sid in the encoded string, which suffices for our attacks to work.

```
DecSid([sk_share^encoded]_{k_M}, [m]_{pk_share}):

    Given: encrypted RSA private key [sk_share^encoded]_{k_M}, encrypted
    message [m]_{pk_share}
    Returns: decrypted and unpadded SID sid'
1  sk_share^encoded ← AES-ECB.Dec(k_M, [sk_share^encoded]_{k_M})
2  N, e, d, p, q, d_p, d_q, u ← DecodeRsaKey(sk_share^encoded)
3  m'_p ← ([m]_{pk_share})^{d_p} mod p
4  m'_q ← ([m]_{pk_share})^{d_q} mod q
5  t ← m'_p − m'_q mod p
6  h ← t · u mod p
7  m' ← h · q + m'_q
8  sid' ← m'[3:45] // Unpad 43 B SID.
9  return sid'
```

Fig. 5. SID decryption during MEGA's client authentication using RSA.

After generating the SID, the server encrypts m with pk_{share} and sends it to the client, together with the encrypted version of sk_{share}.

Figure 5 provides an in-depth description of the client side decryption of the encrypted private RSA key and SID. First, the client decrypts the RSA key, which is encoded for RSA-CRT as follows:

$$sk_{share}^{encoded} \leftarrow \mathsf{I}(q)\|q\|\mathsf{I}(p)\|p\|\mathsf{I}(d)\|d\|\mathsf{I}(u)\|u\|P.$$

Here, q and p are the two 1024-bit prime factors of the RSA modulus N, d is the secret exponent, and $u \leftarrow q^{-1} \bmod p$ is an additional value useful for the RSA-CRT decryption described below. P is padding and $\mathsf{I}(x)$ denotes the two-byte big-endian length encoding of the byte-length of $x \in \{p, q, d, u\}$. For 2048-bit RSA, the encoding consists (with high probability[7]) of three 128 B elements (q, p, and u) and one 256 B part d. Since AES blocks are 16 B, this results in 41 blocks in total, where the last block contains the eight byte padding P.

Next, DecodeRsaKey parses $sk_{share}^{encoded}$ into components and calculates $d_p \leftarrow (d-1) \bmod p$ and $d_q \leftarrow (d-1) \bmod q$. The client only sanity checks the length encodings to ensure that the result is split into exactly four parts. Neither the padding nor the lengths of the individual components are verified. No integrity check is performed during decryption since the key is encrypted using AES-ECB. After decoding the private key, the client performs RSA-CRT decryption of the encrypted SID. Lines 3 and 4 of Figure 5 decrypt the padded session ID m in the smaller rings $\mathbb{Z}_p$ and $\mathbb{Z}_q$. Lines 5–7 recover the padded SID m' using Garner's formula [30]. Instead of checking the padding, line 8 uses the known SID length to truncate m' to sid'. Before truncation, m' is left-padded with zero bytes until its length matches the byte length of N. In a correct execution, we have $sid' = sid$. The client sends sid' in subsequent requests to the server to complete the authentication.

F. MEGAchat

In addition to the cloud storage service, MEGA provides the end-to-end encrypted chat messaging service MEGAchat. The chat messages are encrypted with AES-CTR using 16-byte keys which are periodically rotated. The sender generates these symmetric chat keys and encrypts them for the recipient using the user's long-term asymmetric Curve25519 key. If the Curve25519 public key is not available,[8] the sender uses the RSA-2048 share keys to encrypt the symmetric chat keys as a fallback option.

III. RSA KEY RECOVERY

In this section, we present a practical attack to recover a user's RSA private key by exploiting the lack of integrity protection of key ciphertexts. By tampering with the encrypted RSA private key, a malicious server can deceive the client into leaking information about one of the prime factors of the RSA modulus during the session ID exchange. More specifically, the session ID that the client decrypts with the mauled private key and sends to the server will reveal whether the prime is smaller or greater than an adversarially-chosen value. This information enables a binary search for the prime factor, with one comparison per client login attempt, allowing the adversary to recover the private RSA key after 1023 client logins. The number of required login attempts can be reduced from 1023 to 512 by implementing a lattice-based optimization, which allows an attacker to terminate the search early and recover the remaining missing bits of the prime.

A. Threat Model

We assume a malicious service provider (that is, the adversary controls MEGA's core infrastructure).

B. Attack Description

The attack begins with a *key overwriting* step, in which the attacker exploits the lack of integrity protection of key ciphertexts to modify the client's outsourced RSA private key. The resulting key is altered in the last part of the encoding before the padding, such that it contains $u' \neq u = q^{-1} \bmod p$. When the client uses the thus modified private RSA key to decrypt a ciphertext $[m]_{pk_{share}}$ containing a message m chosen by the adversary, the result leaks information about whether $m < q$ or $m \geq q$, where $q \in [2^{1023}, 2^{1024} - 1]$ is one of the prime factors of the RSA modulus N.

This *case distinction oracle* arises due to properties of RSA-CRT, and allows the adversary to perform a *binary search* for q, by choosing m such that the search interval is halved by each client decryption. To make the client decrypt messages of its choice, the adversary uses the session ID sent from the server at the start of each session. That is, instead of choosing a SID the way the server would, the attacker sets

m to the middle value of the remaining search interval for q and sends $[m]_{pk_{share}}$, the encryption of m, in the place of the encrypted SID. Once one factor q has been determined, the adversary can easily recover the remaining private key.

Next we describe the details of each step of the attack.

1) Key Overwriting: First, $[sk_{share}^{encoded}]_{k_M}$ is modified such that the resulting decrypted RSA private key contains a different u-value than the original private key encoding. Recall that u is a 128 B value spanning blocks 33–41 of $sk_{share}^{encoded}$, with partial coverage of blocks 33 and 41. Since the encoded key is encrypted with AES-ECB, it can be altered at block granularity by modifying the corresponding ciphertext block. Hence the desired modification can be achieved by applying any non-identity transform to one of ciphertext blocks 33–41. Although the resulting value is unknown, it suffices for the attack that the client recovers $u' \neq u := q^{-1} \mod p$. In our implementation of the attack, we modify the second to last ciphertext block of $[sk_{share}^{encoded}]_{k_M}$ to maintain correct length encodings and to avoid garbling the padding. The former ensures that the client's decoding succeeds. The latter increases the robustness of the attack in case client versions that we did not analyze were to perform a format check on the padding.

2) RSA-CRT Case Distinction Oracle: In the second step of the attack, the adversary chooses a message m, encrypts it to $[m]_{pk_{share}}$ and sends to the client in the place of the encrypted SID. When the client uses the modified RSA private key sk'_{share} to decrypt $[m]_{pk_{share}}$, the result allows the attacker to distinguish the case $m < q$ from $m \geq q$. We analyze each case separately to show how the oracle arises. A slight challenge – and notable difference to previous work on fault attacks – is that the adversary only sees part of the result of the decryption, due to the unpadding performed by the client.[9]

Case $m < q$**.** In this case, $[m]_{pk_{share}}$ correctly decrypts to m, despite the fact that the modified key sk'_{share} is used in place of sk_{share}. To see this, first note that if $m < q$, then $m'_q = m$, because by the Chinese remainder theorem (CRT) $m \equiv_q m'_q$, and since $m < q$ we have $m \mod q = m$. For m'_p, we again have by the CRT that $m \equiv_p m'_p$. Therefore, there exists $\alpha \in \mathbb{Z}$ such that $m = m'_p + \alpha \cdot p$. Combining these observations on m'_q and m'_p, we have on Line 5 of Figure 5 that $t \leftarrow -\alpha \cdot p \mod p = 0$. Therefore, $h = 0$, independent of the value of u'. In other words, the client recovers the correct result $m' \leftarrow h \cdot q + m'_q = m$ despite the modified u' value. This results in $sid' = 0$ because $m < q < 2^{1024}$. (Recall that q is a 1024-bit prime.) The client left pads the decrypted m' with zero bytes to 256 B and then removes the rightmost 211 B. Therefore, m is hidden in the padding, and the client recovers and uses $sid' = 0$.

Case $m \geq q$**.** In this case, the message m will not be correctly decrypted, allowing the adversary to detect that the value returned by the client is different from the one sent. To see this, we consider each step of the RSA-CRT decryption procedure again.

By definition, there exist $\alpha, \beta \in \mathbb{Z}$ such that $m'_p = m - \alpha \cdot p$ and $m'_q = m - \beta \cdot q$. Then, $t \leftarrow m'_p - m'_q \mod p = \beta \cdot q \mod p$. Since p and q are coprime, $t \neq 0$ iff $\gcd(\beta, p) = 1$. This happens with overwhelming probability $1 - 1/p$ for primes chosen uniformly at random. Thus, with high probability (w.h.p.), $h \leftarrow t \cdot u' \mod p \neq 0$ and $m' \leftarrow h \cdot q + m'_q \neq m$. Although $m' \equiv_q m'_q$, we have $m' \not\equiv_p m'_p$ because $u' \neq q^{-1} \mod p$ and, therefore, Lines 5–7 of Figure 5 no longer apply the CRT. We observe that $m' \geq 256^{211}$ w.h.p. because of the summand $h \cdot q$. The integers h and q are random numbers of approx. 1024 bits. Therefore, $h \cdot q$ has approx. 2048 bits. Thus, w.h.p., m' is larger than 256^{111}, giving $sid' \neq 0$.

In conclusion, despite the truncated decryption oracle, the adversary can distinguish whether $q < m$ or $m \geq q$ with overwhelming probability based on whether the session ID sid' returned by the client is 0 or not.

3) Binary Search: Using the RSA-CRT case distinction oracle, the adversary can perform a binary search for the RSA prime factor q. For each login attempt by the victim, the adversary chooses m to the middle value of the remaining search interval for q and then uses m instead of the padded SID. Note that due to the lax padding format used by MEGA, clients accept any integer $m \in [0, N-1]$ as a valid padded SID. Once the RSA factor q has been determined this way, the adversary can easily recover the remaining private key as $p \leftarrow N/q$ and $d \leftarrow e^{-1} \mod (q-1)(p-1)$.

C. Impact

A compromised RSA private key allows the adversary to break the confidentiality of all files shared with the victim by other MEGA users. Furthermore, chat keys that are exchanged using the RSA public key as a fallback can be compromised. Of even higher interest than the direct consequences of the recovering the RSA key, however, is the impact of this attack on the overall security of MEGA's key hierarchy. The attacks described in Sections IV and V show how the confidentiality and integrity of user data can be broken by chaining this attack with other vulnerabilities.

D. Complexity and Optimizations

Without optimizations, the attack requires 1023 queries due to the binary search on an interval of the size 2^{1023}. In the MEGA web client, a user needs to perform one login (i.e., enter their password) for every query. If the adversary is the service provider, it can stealthily mount the attack by accepting any SID returned by the victim. In this case, the client may still fail later due to the garbled RSA private key. However, decryption errors caused by the faulty key are not always exposed to the user; on some occasions, the client simply removes and re-fetches the private key.

We briefly discuss how the number of required login requests can be reduced to make the attack faster in practice.

1) Lattice-Based Optimizations: When the most significant bits of the RSA prime factor have been recovered using the technique described above, the remaining bits can be determined without further interaction with the client by using

---

[9]Recall that in an honest execution, the plaintext m contains the padded session ID, which the client truncates to 43 B before returning it to the server.

lattice cryptanalysis. This allows the binary search to be terminated early, requiring fewer login attempts from the user. In Appendix A we describe the straightforward application of a low-dimensional lattice attack adapted from Gabrielle and Heninger [31] to the setting of our RSA key recovery attack. This approach recovers up to 341 unknown bits and therefore reduces the required number of queries for the attack from 1023 to 683. With more complex lattices described by Howgrave-Graham [32] and May [33], it is possible to recover up to 512 unknown bits (i.e., the attack needs only 512 queries).

2) Malicious Client Modifications: MEGA's current web clients store user key material and the SID in the browser's local storage by default. Users remain logged in, and the browser can access this key material without the need for the user to enter their password. We remark that a malicious provider could easily modify current clients to re-establish a new SID in the background instead of storing it locally. With such a seemingly harmless code change, the adversary could perform queries whenever the user accesses the MEGA cloud storage, entirely unbeknownst to the user.

IV. PLAINTEXT RECOVERY ATTACK

As we have seen in the previous section, MEGA's authentication protocol can be used as an oracle to recover a user's private RSA share key sk_{share}, even though it is encrypted with the user's master key using AES-ECB. MEGA encrypts the chat, sign, and node keys in the same manner, reusing the master key and without adding integrity protection for any of the encrypted keys. This leads to the natural question of whether, having recovered sk_{share}, the adversary can go on to also recover the other keys.

In this section, we show that this can be done: after inserting target AES-ECB ciphertext blocks encrypted under the master key k_M into the AES-ECB ciphertext for sk_{share} and choosing the SID in a special way, the session ID returned from the client during authentication leaks the corresponding plaintext blocks. For technical reasons explained below, this method can be used to recover up to two plaintext blocks for each run of the authentication protocol.

A. Threat Model and Adversary

Our threat model considers an adversary that controls MEGA's servers. Additionally, we assume that the adversary knows the client's RSA private key. For instance, it can recover this key by running the RSA private key recovery attack from Section III or performing a forensic investigation of the user's unattended device (e.g., dumping the memory).

The adversary aims to decrypt two (not necessarily consecutive) ciphertext blocks $ct_1, ct_2 \in \{0,1\}^{128}$ that are encrypted with AES-ECB under the master key k_M.

B. Attack Description

The attack consists of three steps: key overwriting, simplifying RSA-CRT and recovering the plaintext.

1) Key Overwriting: Recall from Section II-E the encoding of the RSA private key exchanged during client authentication:

$$sk_{share}^{encoded} \leftarrow \mathsf{l}(q)\|q\|\mathsf{l}(p)\|p\|\mathsf{l}(d)\|d\|\mathsf{l}(u)\|u\|P$$

The client encrypts this key using AES-ECB under the master key k_M before uploading $[sk_{share}^{encoded}]_{k_M}$ to MEGA. The adversary can modify this ciphertext at AES block granularity since AES-ECB encrypts blocks independently. The $[sk_{share}^{encoded}]_{k_M}$ can be split into 41 AES ciphertext blocks $c_1, c_2, \ldots, c_{41}$:

$$c_1\|c_2\|\ldots\|c_{41} \leftarrow [sk_{share}^{encoded}]_{k_M},$$

where $|c_i| = 16$ bytes for all $i \in [1, 41]$. Of particular interest to the attack are the last 9 blocks, which contain the encryption of u.[10] Specifically, block c_{33} encrypts the concatenation of the last 6 bytes of d, the length-encoding $\mathsf{l}(u)$, and $u[1:8]$ (the first 8 bytes of u). Blocks c_{34}–c_{41} contain the ciphertext for $u[9:128]$ including 8 bytes of trailing padding.

For the plaintext recovery attack, we avoid modifying c_{33} to preserve $\mathsf{l}(u)$ and enable successful client-side key parsing. Instead, we overwrite c_{34} and c_{35} with the target ciphertext blocks ct_1 and ct_2. That is, the adversary sends the following tampered encryption of the RSA key to the client.

$$[sk_{share}^{encoded}]'_{k_M} \leftarrow c_1\|\ldots\|c_{33}\|ct_1\|ct_2\|c_{36}\|\ldots\|c_{41},$$

This replacement results in a new RSA private key component, u', which can be split into three parts $u_1\|x\|u_2 \leftarrow u'$, where $u_1 = u[1:8]$, x is the decryption of $ct_1\|ct_2$, and $u_2 = u[41:128]$ is the remaining preserved plaintext bytes of the original u. The adversary aims to recover x, which replaces the 32 bytes $u[9:40]$ of u in u'.

2) Simplifying RSA-CRT*:* To recover x, we leverage that the RSA-CRT decryption of the session ID with the modified sk'_{share} uses u'. By assumption, the adversary knows the original sk_{share}, including u_1 and u_2. By replacing the session ID with a specific message m, the RSA-CRT decryption can be simplified to enable the recovery of x.

The adversary chooses $m \leftarrow u \cdot q$ as the "session ID" and encrypts it using the user's RSA public key.[11] The adversary sends $[sk_{share}^{encoded}]'_{k_M}$ and $[m]_{pk_{share}}$ to the client, which runs the RSA-CRT SID decryption described in Figure 5. The particular choice of m gives

$$m'_p \leftarrow ([m]_{pk_{share}})^{d_p} \bmod p = u \cdot q \bmod p = 1 \text{ and}$$
$$m'_q \leftarrow ([m]_{pk_{share}})^{d_q} \bmod q = u \cdot q \bmod q = 0.$$

This computation is not affected by the modified private key since it only uses p, q, and d. Furthermore, $m'_p = 1$ and $m'_q = 0$ lead to $t = 1$ and $h = u' \bmod p$. Consequently, the decryption of $[m]_{pk_{share}}$ simplifies to $m' = h \cdot q$.

We now argue that with high probability, $m' = u' \cdot q$. If the

---

[10]In the following, we focus on the case where the private exponent d is 256 bytes and u is 128 bytes to simplify the analysis. This means that u spans blocks 33–41. It would be straightforward to adapt our attack to corner cases with shorter encodings of d or u.

[11]Although m does not have the expected form of a padded SID, the client-side processing tolerates any message (cf. Figure 5).

adversary were given m', it would be easy to recover u', and thereby x, by computing $u' \leftarrow m'/q$. We discuss later how the adversary can still recover x although it only receives 43 bytes of m'. First, for $m' = u' \cdot q$ to hold, we need that $u' \bmod p = u'$, i.e., $u' < p$. To see that this is the case w.h.p., recall that $u < p$ by definition. Split the prime p into $p_1 \| p_2 \leftarrow p$, where $|p_1| = |u_1| = 8$ B and $|p_2| = 120$ B. By construction, u and u' both start with u_1. Since $u < p$, it can only be the case that $u' \geq p$ if $u_1 = p_1$. Thus, we derive the following lower bound on the probability.

$$\Pr[u' < p] \geq 1 - \Pr[u_1 = p_1] = 1 - \frac{p_2 + 1}{p} \geq 1 - 2^{-63}$$

Hence, $m' = u' \cdot q$ with probability at least $1 - 2^{-63}$.

3) Recovering Plaintext: We show how an adversary can recover x from the truncated SID $m'[3:45]$. We assume that $u' < p$ such that $m' = u' \cdot q$. Let $y_1 \| y_2 \| y_3 \leftarrow m'$, where y_1 is the removed 2-byte prefix, y_2 is $m'[3:45]$, and y_3 is a 211-byte unknown suffix. To recover x, the adversary tries all possible prefix values $\hat{y}_1 \in \{0,1\}^{16}$ and performs arithmetic operations on $\hat{y}_1 \| y_2 \| y_3$ to get $u_1 \| x$. The correct prefix guess $\hat{y}_1^*$ can be detected when the result starts with u_1.[12]

To compute $u_1 \| x$, interpret the involved byte strings as big-endian encoded integers. Then, from $m' = u' \cdot q$, $m' = \hat{y}_1^* \| y_2 \cdot 256^{211} + y_3$, and $u' = u_1 \| x \cdot 256^{88} + u_2$, we obtain

$$\frac{\hat{y}_1^* \| y_2 \cdot 256^{211}}{q} = u_1 \| x \cdot 256^{88} + u_2 - \frac{y_3}{q}.$$

The last term is bounded by $\frac{y_3}{q} < 2^{665}$. Therefore, at least 39 bits separate the prefix $u_1 \| x$ from $\frac{y_3}{q}$. Thus, the subtraction of $\frac{y_3}{q}$ can only affect x if u' has the prefix $u_1 \| x \| 0^{39}$. This happens with a probability of about 2^{-39} since u' is approximately uniformly distributed. Hence, with probability $1 - 2^{-39}$,

$$u_1 \| x = \left\lfloor \frac{\hat{y}_1^* \| y_2 \cdot 256^{211}}{q} \cdot 256^{-88} \right\rfloor,$$

where u_2 is rounded away because it is smaller than 256^{88}.

C. Complexity

The adversary recovers the two plaintext blocks (32 B) corresponding to ct_1, ct_2 with a probability of at least $1 - 2^{-39}$, given that $u' < p$. Let S be the event that the attack is successful. Then, the overall success probability is

$$\Pr[S] = \Pr[S \mid u' < p] \cdot \Pr[u' < p] > 1 - 2^{-38}.$$

The adversary needs to iterate over the 2^{16} prefixes $\hat{y}_1$, which is computationally trivial and does not involve any interaction with the victim. Additionally, a single login attempt of the user is required to perform the attack. Since each successful instantiation of the attack recovers two AES-ECB plaintext blocks, one query suffices to recover a full node key or 32

---

[12]This method has a small false positive probability of approximately 2^{-64} (since $|u_1| = 64$). However, this can be avoided in practice using additional information to detect when the the correct plaintext x has been recovered (e.g., by attempting to decrypt a file using x as a key).

---

bytes of asymmetric key material. The attack cannot be used to decrypt three or more blocks per login attempt because then the prefix $u_1 \| x$ is changed by the unknown term $\frac{y_3}{q}$ with high probability. However, the adversary can iterate the attack to recover as much plaintext as it desires.

D. Practical Considerations

In practice, the web client often executes a special case[13] (not shown in Figure 5), where only a single prefix byte is removed during SID decryption instead of two. The above attack and analysis are straightforward to adapt to this case. The adversary iterates over the 2^8 values $\hat{y}_1 \in \{0,1\}^8$ and recovers the prefix $u_1 \| x$ with probability $1 - 2^{-31}$. The overall success probability is $\Pr[S] > 1 - 2^{-30}$, and the computational cost is slightly lower.

V. Integrity Attacks

Having successfully recovered the node, share, chat, and sign keys of any MEGA user using the attacks in the preceding sections, it is clear that at this point all confidentiality of user data is lost. We now turn our attention to integrity, to see what guarantees MEGA's system gives users in terms of file authenticity. The result is – perhaps unsurprisingly – that after recovering node keys, very little protection remains: access to the keys means that the adversary can trivially *modify* existing files by decrypting, changing, and then re-encrypting the files. More interesting is the ability to add new files to the user's storage, without relying on existing node keys. We present two versions of this stronger type of attack.

In the first, an attacker can create a node key ciphertext by choosing any two AES-ECB ciphertext blocks, and use the plaintext recovery attack from Section IV to decrypt the obfuscated key. It may then use the knowledge of the resulting key, nonce and metamac to forge a valid file ciphertext for a plaintext of its choosing, up to one AES block. This enables a *framing attack* on the victim, who will not be able to provide cryptographic evidence that they did not upload the forged file.

The second attack exploits the structure of MEGA's obfuscated node keys to create a key ciphertext for the all zero key: by repeating a ciphertext block the adversary can ensure that the client derives the key $k_F = 0^{128}$. This attack is less surreptitious than the framing attack because of the low probability of the all-zero key appearing in an honest execution. In return, it does not rely on the ability to decrypt arbitrary AES-ECB ciphertexts; a single known plaintext-ciphertext AES block pair suffices. With a known key, the attacker can forge a valid ciphertext for a chosen plaintext.

A. Threat Model

The threat model considers an adversary controlling MEGA's core infrastructure. The first attack assumes access to a decryption oracle $\mathcal{O}_{dec}$ for AES-ECB encryption under the master key k_M. This oracle can be realized by exploiting the attack in Section IV, for example.

---

[13]This special case is unlikely to occur during normal operation but happens with probability $1 - 2^{-8}$ for our choice $m = u \cdot q$ for the SID.
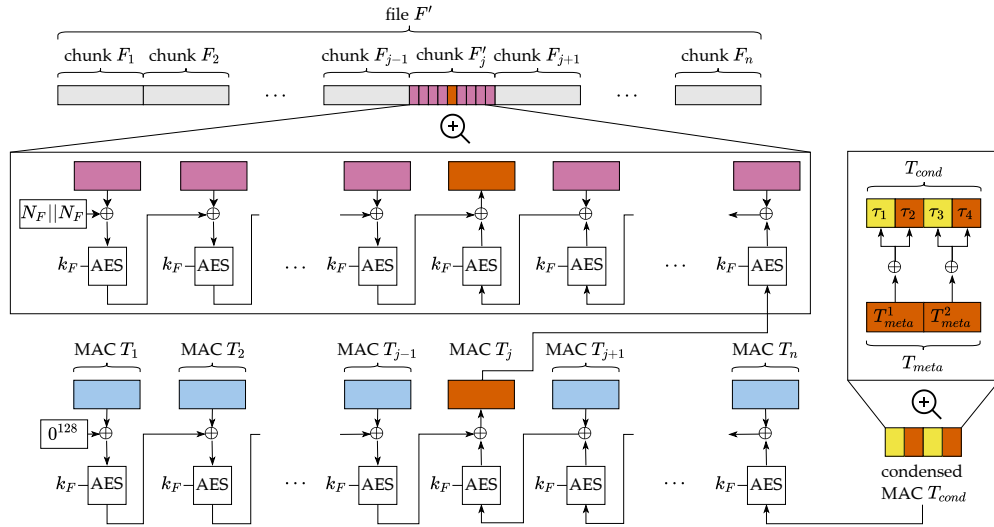
Fig. 6. Reconstruction of a mostly chosen file using a known node key k_F and nonce N_F that produces a fixed metamac T_{meta}.

The second attack requires knowledge of a single plaintext-ciphertext pair (pt, ct) such that $ct = \mathsf{AES\text{-}ECB.Enc}(k_M, pt)$. This can again be obtained from the plaintext recovery attack in the preceding section, but can also be acquired in other ways. For instance, the attacker can use MEGA's protocol for public file sharing to obtain the pair. When a user shares a file or folder publicly, they create a link containing the obfuscated node key in plaintext. Hence, a malicious cloud provider who obtains such a link knows both the plaintext and the corresponding ciphertext, since the latter is uploaded to MEGA when the file is created (before being shared).

B. Attack Description

We first describe the two approaches to derive key material for the file forgery, based on the assumed adversarial resources. Apart from how the key is derived, both attacks then use the same procedure to construct a file and corresponding ciphertext which will pass all integrity checks.

1) Decryption Oracle: In this scenario, the adversary has access to an AES-ECB decryption oracle $\mathcal{O}_{dec}$. To create a key ciphertext for which the attacker knows the plaintext obfuscated key, it proceeds as follows. The adversary selects two AES ciphertext blocks $ct_1, ct_2 \in \{0,1\}^{128}$ uniformly at random. Next, it uses the decryption oracle to recover the corresponding plaintext blocks $pt_1 \| pt_2 \leftarrow \mathcal{O}_{dec}(ct_1 \| ct_2)$. It then runs the key deobfuscation algorithm described in Figure 4 to obtain the node key, nonce, and metamac: $k_F, N_F, T_{meta} \leftarrow \mathsf{DeobfKey}(pt_1 \| pt_2)$. Note that because of the random choice of ciphertext blocks, the resulting encryption key, nonce, and tag are indistinguishable from those of a genuinely uploaded file.

2) Single Plaintext-Ciphertext Pair: In this second scenario, we assume that the adversary knows a plaintext-ciphertext pair (pt, ct) where $ct = \mathsf{AES\text{-}ECB.Enc}(k_M, pt)$. Given this, the adversary can forge a key ciphertext that decrypts to a node key of all zero bytes. The forgery is

possible due to the structure of obfuscated keys combined with AES-ECB encryption. The adversary chooses $ct \| ct$ as encryption for the obfuscated file key. By construction, this key ciphertext decrypts to $k_F^{obf} = pt \| pt$, since AES-ECB decrypts the two blocks independently and with the same key k_M. For $k_F, N_F, T_{meta} \leftarrow \mathsf{DeobfKey}(pt \| pt)$ this gives $N_F \| T_{meta} = pt$ and $k_F = pt \oplus (N_F \| T_{meta}) = 0^{128}$.

Note that this works regardless of the plaintext content of the AES blocks. Hence the attacker can use any plaintext-ciphertext pair (pt, ct). The decryption and deobfuscation of the key ciphertext will always succeed since node keys are not integrity-protected.

3) File Reconstruction: The adversary now has a node key k_F, a nonce N_F and a metamac T_{meta} and wants to forge a ciphertext for a file F such that it verifies under the tag T_{meta}. On a high level, the adversary achieves this by working backward from the metamac and inserting a single AES block at a convenient location in the malicious file F. Note that many standard file formats such as PNG and PDF tolerate 128 injected bits (for instance, in the file's metadata, as trailing data, or in unused structural components) without affecting the displayed content. Hence the modified file F' can be constructed to appear identical to F.

Figure 6 visualizes the construction of F' given k_F, N_F, and T_{meta}. The dark orange blocks are fixed and imply constraints that must be satisfied for the forged file to pass the integrity verification. The adversary starts by creating a condensed MAC T_{cond} that produces T_{meta}. For this purpose, it splits T_{meta} into two 32-bit chunks T_{meta}^1 and T_{meta}^2. Next, it chooses $\tau_1, \tau_3 \leftarrow_\$ \{0,1\}^{32}$ u.a.r. and sets $\tau_2 \leftarrow \tau_1 \oplus T_{meta}^1$ and $\tau_4 \leftarrow \tau_3 \oplus T_{meta}^2$. This ensures that the condensed MAC $T_{cond} \leftarrow \tau_1 \| \tau_2 \| \tau_3 \| \tau_4$ produces T_{meta} when the metamac is computed.

Next, the file chunk MAC tags must be set to ensure that the condensed MAC is T_{cond}. Let $F_1 \| F_2 \| \ldots \| F_n \leftarrow F$ be the

file chunks of the adversarially chosen file, each consisting of l_F AES blocks. Recall from Figure 2 that T_{cond} is the CBC-MAC of the concatenation of all n chunk MACs T_i for $i \in [1, n]$. Because of the structure of CBC-MAC, all but a single chunk MAC tag can be chosen freely to give the desired T_{cond}. The last tag can then be reconstructed from the other tags and T_{cond}. Hence, to proceed, the adversary selects a chunk index $j \in [1, n]$ such that the file format of F tolerates 128 random bits (aligned to AES blocks) in the j-th chunk. Then, it calculates all chunk MACs except T_j using MEGA's AES-CCM* encryption (cf. Figure 3):

For all $i \in [1, n] \setminus \{j\}$ do :
$$[F_i]_{k_F}, T_i \leftarrow \mathsf{AES\text{-}CCM^*.Enc}(k_F, N_F \| (i \cdot l_F), F_i)$$

Next, the adversary computes the condensed MAC tag for chunk j by applying a "meet-in-the-middle" CBC-MAC calculation, to ensure that the result of CBC-MAC over all the chunk MACs is T_{cond}. That is, it computes the intermediate condensed MAC $T_{cond,j-1}$ up to chunk j (the "forward direction") by calculating the CBC-MAC of $T_1 \| T_2 \| \ldots \| T_{j-1}$. It also computes the intermediate condensed MAC values for chunks $j + 1$ to n backward, starting from the desired output $T_{cond,n} \leftarrow T_{cond}$. That is, for $i = n - 1, n - 2, \ldots, j$ let

$$T_{cond,i} \leftarrow \mathsf{AES\text{-}ECB.Dec}(k_F, T_{cond,i+1}) \oplus T_{i+1}.$$

The remaining tag T_j is defined by $T_{cond,j-1}$ and $T_{cond,j}$:

$$T_j \leftarrow T_{cond,j-1} \oplus \mathsf{AES\text{-}ECB.Dec}(k_F, T_{cond,j})$$

Now, the MAC tag T_j for file chunk j is fixed. For analogous reasons as above, the adversary can construct and insert a single AES block into F_j such that the MAC of the resulting file chunk F'_j is T_j. The adversary then sets

$$F' \leftarrow F_1 \| \ldots \| F_{j-1} \| F'_j \| F_{j+1} \| \ldots \| F_n$$

and generates the file ciphertext C_F by encrypting the file chunks with the key k_F. The adversary can also encrypt file attributes of its choice using AES-CBC with key k_F. Note that any attributes may be chosen and that no modification is necessary since file attributes are not integrity-protected by MEGA. Finally, the attacker places the key ciphertext (either $ct_1 \| ct_2$ for the two randomly chosen ciphertext blocks in the decryption oracle scenario, or $ct \| ct$ for the known plaintext-ciphertext pair (pt, ct)), the file ciphertext and the attribute ciphertext in the victim's cloud storage.

C. Impact

With either attack, the adversary is able to add a new file to the user's cloud. The file can be chosen by the adversary, up to one AES block in a flexible location. The impact of this fixed block is small in practice since many file formats tolerate sufficiently long sections of arbitrary bytes.

In MEGA's threat model, the expected file integrity protection is that only the user can upload files to their storage due to the client-side user-controlled encryption. Hence an adversary exploiting these vulnerabilities can frame a user for possession of incriminating files, that, in theory, only the victim could be the creator of. For example, a conceivable attack might frame someone as a whistle-blower and place an extensive collection of internal documents in that person's account.

D. Complexity

The attacks require only a trivial amount of computation. The file reconstruction solely uses simple bit operations and fast AES block cipher applications. If the decryption oracle is instantiated with the plaintext recovery attack from Section IV, the attack needs a single user login attempt. The second integrity attack does not require additional effort.

VI. GUESS-AND-PURGE BLEICHENBACHER ATTACK

In this section, we present a new Guess-and-Purge variant of Bleichenbacher's attack [4] (GaP-Bleichenbacher) to decrypt RSA ciphertexts using a padding oracle exposed by the fallback chat key exchange for MEGAchat. The attack is adapted from PKCS#1 v1.5 padding to the custom padding scheme used by MEGA clients for RSA encryption of chat keys when no Curve25519 key is available. Our attack devices a new strategy that *guesses* the unknown two-byte prefix tolerated by this padding scheme and quickly *purges* wrong guesses. The overall GaP-Bleichenbacher attack requires $2^{16.9}$ client login attempts on average to decrypt one ciphertext.

Although this attack is weaker than the RSA key recovery in Section III (in the sense that a key recovery implies plaintext recovery), it is complementary in the vulnerabilities that it exploits and hence requires separate countermeasures. Additionally, the Bleichenbacher attack may be performed by a weaker adversary.

A. Threat Model and Adversary

The threat model assumes an adversary with chosen-plaintext capability for the RSA encryption used as a fallback chat key exchange. The adversary needs a channel to the victim over which it can send encrypted chat keys. The client throws different errors depending on whether the RSA decryption of the chat keys was successful or not. We assume that the adversary is able to observe this error oracle.

We outline two possible realizations of this adversary. First, a malicious service provider can send any message encrypted with the target's RSA public key to the user disguised as encrypted chat keys. The client reports a different error message to the server when RSA decryption fails and when chat message decryption fails (because random bytes are used as chat key after a successful RSA decryption).

Second, another user who has a direct chat with the victim may execute the same attack by sending maliciously chosen messages instead of chat keys during the key exchange. We consider it possible that such an adversary can infer the target's decryption success. Error messages that the target sends to the server may be forwarded to the chat partner to inform the sender that transmission failed. Otherwise, the adversary could observe the encrypted network traffic between MEGA and the target's client and distinguish error messages sent to the server based on timing and message sizes.

B. Attack Outline

Recall that Bleichenbacher's attack [4] on PKCS#1 v1.5 padding maintains an interval of possible plaintexts. It exploits the malleability of RSA to test the decryption of multiples of the unknown target message. Successful unpadding leaks the prefix of the decrypted message due to the structure of the PKCS#1 v1.5 padding. This prefix allows an adversary to reduce intervals efficiently and recover the plaintext.

MEGA's padding scheme has an unknown prefix of two bytes that prevents the direct application of Bleichenbacher's attack. Every successful decryption corresponds to many disjoint solution interval candidates leading to a state explosion.

Our attack guesses the unknown prefix and removes it before updating the solution interval. We find empirically that wrong guesses lead to an empty solution interval within a few iterations of the GaP-Bleichenbacher attack steps. By using practical optimizations – including dynamic programming and early termination – we can avoid both repeating queries and spending time to recover padding bits.

We provide a detailed description of GaP-Bleichenbacher, including the intricate adaption of Bleichenbacher's attack steps, in Appendix B.

C. Complexity

Our experiments evaluated the attack to require $2^{16.9}$ queries on average, where 25% of all runs need less than 2^{14} queries, and the distribution has a long tail. We provide further details in Appendix B.

D. Impact

An adversary can use a vulnerable client to decrypt any RSA ciphertext due to the reuse of a single RSA key pair. Hence, the attack enables the recovery of chat keys transferred using RSA, as well as node keys shared with the victim.

VII. Proof of Concept

We set up a test account on MEGA and implemented a PoC of our attacks to test them in practice.[14] Each PoC is implemented in two settings: `sim` and `real`.

In `sim`, the entire attack is run against a local simulation of the relevant parts of MEGA to avoid affecting MEGA's operation. In `real`, we verify that our simulation accurately models MEGA's system by carefully testing the components of the attack on the MEGA web client v. 4.11.2.[15] Since the server code is not published, we cannot implement a PoC where the adversary controls MEGA. Instead, we implement a MitM attack by installing a bogus TLS root certificate on the victim. This setup allows us to impersonate MEGA towards the user while using the real servers to execute the server code (which is unknown to us). We can patch server responses and perform our attacks on the fly since they do not rely on secrets stored by the server.

[14]The implementation of our PoCs is published on GitHub: https://github.com/MEGA-Awry/attacks-poc.

[15]Since we exploit fundamental flaws in the cryptographic architecture, we expect the vulnerabilities to apply to other clients and versions as well.

Fig. 7. Forged file with 128 chosen bits after IEND, the last PNG chunk.

For our RSA key recovery attack from Section III, we ran the full binary search and simple lattice optimization described in Appendix A in `sim` to ensure that the attack succeeds reliably in 683 login attempts. The lattice recovery of 341 missing bits of the prime succeeded in 1000 out of 1000 runs. Afterwards, we recovered the first and last few bits of the RSA prime factor in the `real` setting. This way, we were able to verify the correctness of the attack while avoiding having to perform excessively many login requests on MEGA's servers.

Our plaintext recovery attack from Section IV only requires a single login query to recover a node key when the RSA private key is known. Therefore, we implemented a full PoC in both settings. We investigated the internal state of MEGA's command-line client to obtain the private key material of our test account and the expected node key decryption.

Using the integrity attacks from Section V, we successfully forged a valid ciphertext for the PNG image shown in Figure 7. The PNG file format ignores any data appended to the image, making it simple to modify the image to add the necessary block needed to pass the integrity verification. In `sim`, we verified that our reconstructed MEGA file decryption successfully recovered the forged file. For `real`, we implemented the attacks only on the client-side to avoid uploading persistent bogus material to MEGA. We injected the file in the file tree hierarchy fetched by the web client and then intercepted the load request caused by the user when opening our forged image in the MEGA image viewer. Then, we served our forged file and verified that it displayed correctly on the client.

For the fallback chat key transfer over RSA that the GaP-Bleichenbacher attack from Section VI targets, source code comments suggest that it is only used for accounts registered before 2016. Downgrade attacks for newer accounts cannot be ruled out but we did not attempt to implement this attack in the `real` setting due to the closed-source server code and the substantial number of queries. Instead, we only implemented it in `sim` to show that MEGA's custom padding scheme is fundamentally vulnerable to an adaption of Bleichenbacher's attack on PKCS#1 v1.5 padding.

VIII. Countermeasures

As part of our disclosure to MEGA, we detailed three sets of countermeasures: *immediate* patches, suggesting backward-compatible mitigations to temporarily protect against the most severe consequences of our attacks, *minimal* patches, providing more robust protection while avoiding expensive operations like re-encryption of all user files, and *recommended* measures, proposing steps toward a redesign of MEGA's cryptographic architecture. Figure 8 shows an overview of the proposed changes to the key hierarchy for each set of mitigations.

A. Immediate Countermeasures

1) Integrity-Protect Key Ciphertexts: The most effective countermeasure against our attacks is to add integrity protection for the encrypted user keys stored by MEGA. This can be done in a non-invasive way by adding HMAC tags to the key ciphertexts. By extending the existing encryption, older clients can ignore the new authentication tags and remain functional. We advise the use of distinct keys for separate usages of HMAC, rather than re-using the master key, to avoid further vulnerabilities from the lack of key separation. We refer to a key used for this purpose as a "key integrity key" (KIK) in Figure 8.

This measure directly protects against the RSA key recovery in Section III. Consequently, it also prevents our plaintext recovery and integrity attacks (Sections IV and V) because they build on the RSA key decryption and rely on the lack of key ciphertext intergrity. However, this measure should only be considered a temporary patch, since AES-ECB-then-HMAC still does not achieve authenticated encryption security. Nevertheless, we propose this as a temporary solution due to MEGA's challenging scale, the urgency of the issues, backward compatibility considerations, and the ease of implementation.

2) Separate Keys: MEGA broadly violates the *principle of key separation*: the practice of using separate keys for separate purposes. The most notable instance is the reuse of the master key to encrypt all other user keys, enabling the AES-ECB plaintext recovery attack in Section IV. As an immediate measure, we propose to replace the master key with a new, randomly chosen key derivation key, k_D, and to use HKDF to derive a set of *key encryption keys* (KEKs) from k_D to separately encrypt the share, chat, sign, and node keys.

This measure offers additional protection against the AES-ECB plaintext recovery attack: the RSA private key decryption can no longer be used to decrypt other encrypted keys, since they are encrypted with distinct KEKs. However, users should also change their passwords after this patch is implemented, as a proactive measure to render the old master key k_M inaccessible. Without a password change, the encryption key k_e remains the same and can still decrypt k_M. Thus, if a user could be tricked into decrypting the master key ciphertext (for instance, by using an outdated client), our attacks could still be performed by a malicious entity that stored earlier versions of key ciphertexts. Nevertheless, even users who do not update their password would still benefit

from the proposed key separation, as the AES-ECB plaintext recovery attack could no longer be used to compromise the keys of newly uploaded files.

3) Use a Stricter RSA Padding Format: We propose to enforce stricter client-side checks on MEGA's custom RSA padding to increase the number of queries needed for the GaP-Bleichenbacher attack. Enforcing a fixed 2-byte padding prefix would increase the number of queries needed for the attack to approximately 2^{33} because conforming messages are then harder to find. This modification is a short-term measure that does not remove the padding oracle. An attack requiring 2^{33} queries is still worrisome as it could be improved. Nevertheless, this measure would reduce the practicality of the attack and make it easier to detect.

B. Minimal Countermeasures

The minimal countermeasures address all of our attacks pragmatically. We propose to switch to more standard cryptographic primitives when it does not involve the re-encryption of large data volumes. In particular, this means that old clients ideally need to be deprecated, or, alternatively, that new and old client versions should be supported in parallel. The latter requires carefully designed protocols to avoid downgrade attacks.

1) AES-GCM for Key Ciphertexts: In the long term, a standardized AEAD scheme should be used to encrypt user keys. Hence we propose to replace the ad hoc AES-ECB+HMAC construction introduced in the immediate countermeasures with AES-GCM for the encryption of the private share, chat and sign keys as soon as possible. Ideally, the node keys should also be encrypted with AES-GCM, but we postpone this switch to the recommended countermeasures due to the conceivable complexity of such an operation.

This measure addresses our attacks more adequately than the immediate measures do. It also simplifies the key hierarchy (see Figure 8) by removing all KIKs (except the one for node keys). However, AES-GCM needs to be used carefully to avoid issues with nonce reuse [34], cache side-channel attacks [35], [36], fragile authentication [37], [38] and attacks from lack of key commitment [13], [14], [15]. We advise MEGA to follow standard key-wrapping practices [39] and to use the key's purpose and, for asymmetric primitives, the public key as associated data to authenticate control data and avoid key confusion attacks.

2) RSA-OAEP and separate RSA Keys: We suggest the use of RSA-OAEP [40] for RSA encryption to protect against Bleichenbacher-style attacks on the padding. We further recommend adding an additional RSA key pair $(sk_{share}^{legacy}, pk_{share}^{legacy})$ for the legacy chat key exchange to separate the two uses of RSA encryption. For compatibility reasons, MEGA's custom padding scheme may need to be preserved in this legacy code, in which case this part of the code will still be vulnerable to the GaP-Bleichenbacher attack.

C. Recommended Countermeasures

In this last set of measures, we discuss long-term goals for a cryptographic refactoring of MEGA's architecture to
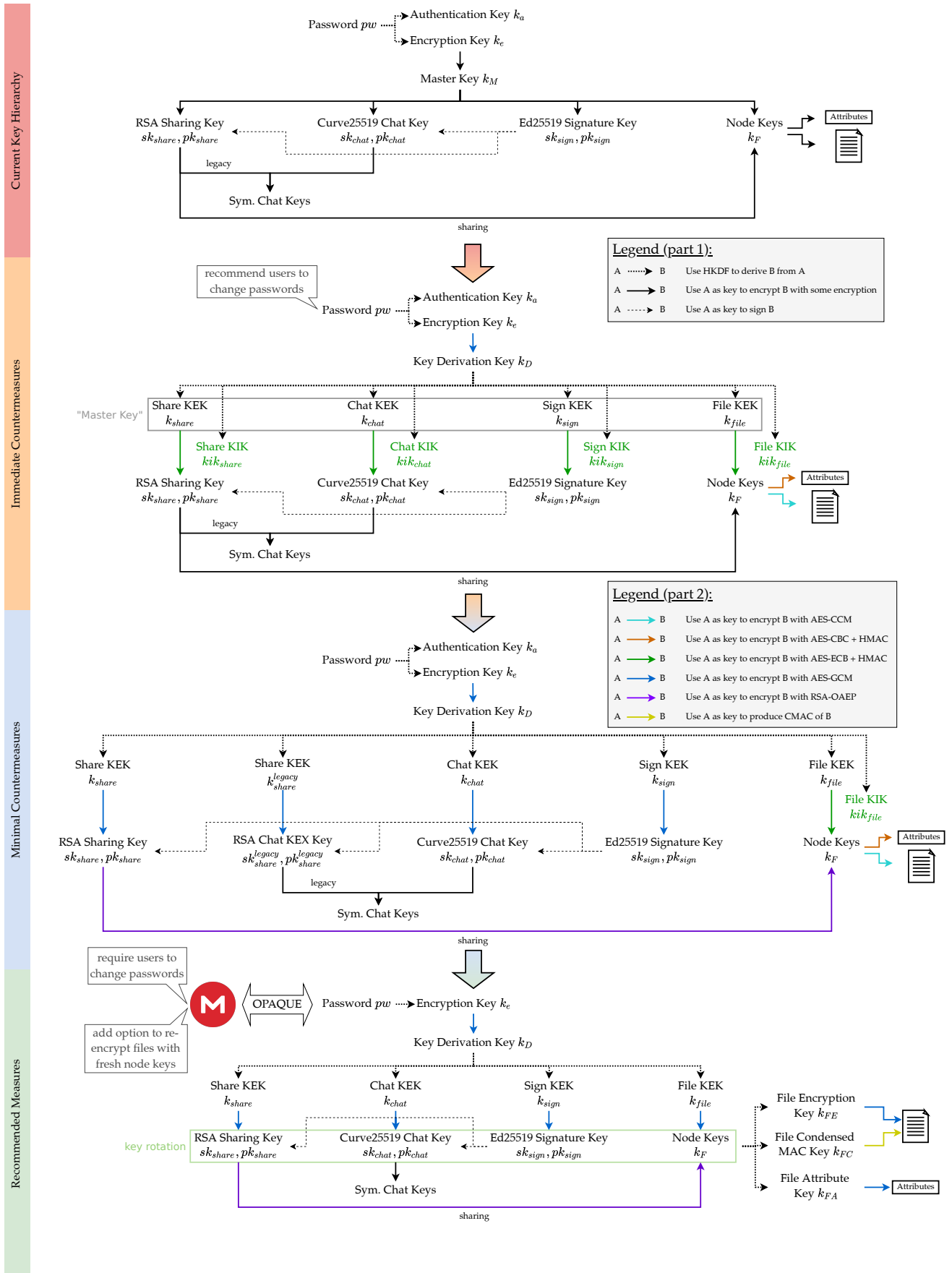
Fig. 8. Redesign of MEGA's key hierarchy for the immediate, minimal, and recommended countermeasures.

adequately address all of our attacks and protect against future ones. The measures proposed in this section no longer retain backward-compatible but insecure functionality. For instance, we propose to remove the legacy chat key transfer and, thus, require users of deprecated clients to update. However, apart from being disruptive, the measures stay in line with the current cryptographic architecture and propose a secure way to implement it without more changes than necessary to the design or functionality.

1) File Key Encryption: Our general recommendation is to use authenticated encryption with associated data to protect user keys (and files), and to use distinct keys for distinct purposes. In particular, when the design of the system changes to introduce updates or new features, new keys should be used for new functions to avoid vulnerabilities from legacy code.

Concretely, we suggest to replace the ad hoc integrity protection of node key ciphertexts introduced in the immediate countermeasures by AES-GCM. Additionally, instead of using the node keys directly, we recommend using HKDF to derive separate keys for file encryption, file attribute encryption and to compute the condensed MAC tag. We also advise against the use of MEGA's "obfuscated" node key format. Lastly, we recommend regular key rotation for all key-encryption keys.

2) File Encryption: As previously noted, the file encryption scheme used by MEGA is a variant of AES-CCM which does not conform to the standard [27] since it does not encrypt the authentication tag. We again propose to replace it with the more widely adopted and efficient AES-GCM. Furthermore, we advise to use CMAC [41] to compute MACs (such as the metamac) over variable-length messages.

3) Augmented PAKE for Authentication: As noted in Section I-B, MEGA implements an unusual authentication procedure that is susceptible to dictionary attacks: The client transmits the authentication key k_a, which is half of the output of PBKDF2-HMAC-SHA512 on the user password, to the server. A MitM adversary capable of breaking TLS can observe k_a and the salt, and try different passwords to find a match. Although publishing a partial output of PBKDF2 is not a violation of the PKCS#5 v1.2 standard, [42] showed that such attacks on PBKDF2 are feasible on custom hardware and GPUs.

We propose to replace the current authentication with OPAQUE [43], an augmented Password Authenticated Key Exchange (PAKE) protocol recommended by the Crypto Forum Research Group [44]. This construction removes the need for an authentication key in MEGA: a user can directly authenticate by proving their knowledge of the password to the server. Meanwhile a MitM adversary capable of breaking TLS cannot gain any advantage in recovering user passwords, even through carrying out pre-computation prior to a server compromise. Note, however, that since our threat model considers the MEGA server to be adversarial, the version of OPAQUE in [43] in which the server does *not* learn the user password during registration should be used. Note also that even using this version, MEGA can still mount dictionary attacks against individual user passwords, by testing password guesses against server-side data needed in OPAQUE.

After implementing OPAQUE, MEGA should request users to change their password. Otherwise an adversary that stored k_a from a previous authentication can still perform a dictionary attack.

IX. DISCUSSION

MEGA is not the first – and almost certainly not the last – system to contain critical security vulnerabilities. While security analysis like ours will remain important in establishing and improving the privacy of users of cryptographic systems, it is unfortunate that such attacks continue to be discovered. When a system has grown popular enough to attract the attention of independent researchers, skilled adversaries may have already compromised the system. Mitigating attacks cannot undo the consequences of such compromises. Additionally, the process to patch a large-scale system like MEGA is at best cumbersome, at worst impossible.

The problem of bridging the knowledge gap between cryptographers and implementers is a long-standing one, and beyond the familiar advice to stick to well-tested implementations of standardized and provably secure primitives, we will not discuss it further in general here. Rather, we choose to highlight some specific lessons learned from our review of MEGA and give recommendations for the future.

A. How and Why MEGA's Design Fails

The attacks presented in this work arise from unexpected interactions between seemingly independent components of MEGA's cryptographic architecture. They hint at the difficulty of maintaining large-scale systems employing cryptography, especially when the system has an evolving set of features and is deployed across multiple platforms. The challenges involved in a complete redesign of a cryptographic architecture can make ad hoc fixes and short-term solutions attractive. In turn, this can lead to even more complexity that becomes more difficult to maintain, due to the introduction of new dependencies and the desire to provide backward compatibility.

As an example, our recommended transition to authenticated encryption in MEGA would require all customers to download, decrypt, re-encrypt, and upload all their data, due to the end-to-end security features of the system. With 1000 PB of data stored by MEGA, this would take more than half a year at MEGA's peak bandwidth of 1000 Gbit/s. It would also place an immense load on MEGA's storage infrastructure. Perhaps because of challenges like this, MEGA decided to deploy a more short-term approach, leading to additional complexity.

Hence, a design that anticipates cryptographic updates and allows new features to easily be added without introducing cross-domain vulnerabilities is crucial. A core feature of such a design is good key hygiene. Careful key separation not only minimizes the risk for unintended and dangerous interactions between cryptographic components, but can also protect against downgrading attacks and vulnerabilities in legacy code.

Another central issue in MEGA's design is the lack of consistent provision of integrity for ciphertexts: MEGA attempted to provide integrity for stored files, but not for the

keys used to protect those files. This gave rise to a complete breach of confidentiality of user data. We hypothesize that this distinction in how integrity is provided arises from a misunderstanding concerning the strength of the relevant threat model for the analysis of MEGA. By now it seems to be well-understood that both confidentiality and integrity are needed when securing *data* at rest, but perhaps this is not so obviously true when securing *keys* at rest when faced with a malicious service provider. Some practitioners may also have the impression that security notions for authenticated encryption assume unrealistically powerful adversaries. However, as our attacks on MEGA show, (partial) decryption oracles can exist in practice, especially in the setting of a malicious service provider. We observe that RSA-CRT is particularly vulnerable to key overwriting attacks in a chosen-plaintext setting since the decryption directly uses the prime factors of the RSA modulus, potentially leaking useful information for factorization. Here, establishing the use of AEAD as the default is an important step in reducing the potential for attacks.

B. Consequences

Besides the effort and computational power required to patch a large system, vulnerabilities like the ones presented in this paper can have dire consequences for the system's users. The attacks presented here show that it is possible for a motivated party to find and exploit vulnerabilities in real world cryptographic architectures, with devastating results for security. It is particularly concerning that services like MEGA – which advertise privacy as a core feature and hence particularly attract users in need of strong protection – fail to withstand cryptanalysis. It is conceivable that systems in this category attract adversaries who are willing to invest significant resources to compromise the service itself, increasing the plausibility of high-complexity attacks. Moreover, the cost of finding and exploiting such vulnerabilities is amortized by the large number of accounts to which they can be applied.

Once the system has been compromised, recovering security (and trust) may be challenging even if the vulnerability is discovered and countermeasures applied. Ideally, users should be able to regain security after a compromise by updating their key material, for example by resetting their password. However, even with such defensive mechanisms in place, in-depth insight into the vulnerability would be needed for users to assess what security guarantees remain, and how well patches protect their old and new data. Of course, we cannot expect consumers to make such assessments, and they may very well lose trust in the provider or fail to (for example) reset their password because they cannot judge the security implications.

C. Future Work

Given the popularity of cloud computing in general, and outsourced storage in particular, it is safe to assume that the demand for secure and private cloud services will continue to rise. Rather than leaving the task of designing a secure system to individual providers – which, paradoxically, would require users to trust the (by assumption) untrusted cloud provider

with a secure design and correct implementation – we advocate for a standardization of secure cloud storage.

Such a standard would ideally provide a secure and robust foundation obviating the need for ad hoc designs, while still leaving room for vendor-specific customization and improvements. For instance, support for additional features should be provided by design, and the specification should be cryptographically agile. Developing a good standard would require deep engagement with a broad spectrum of stakeholders, including but not limited to cryptographers. We believe that this would be the easiest path to avoid attacks stemming from the lack of expert knowledge among developers, and that it would enable users to finally have confidence that their data remains just that – theirs.

REFERENCES

[1] "About Us – Encrypted Cloud Storage – MEGA," https://mega.io/about, visited on March 16, 2022.

[2] "Eight years of MEGA – Tweet," https://twitter.com/MEGAprivacy/status/1352564229044277248?s=20, January 2021, visited on January 10, 2022.

[3] "Security and Why It Matters – MEGA," https://mega.io/security, visited on March 31, 2022.

[4] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1," in *CRYPTO'98*, ser. LNCS, H. Krawczyk, Ed., vol. 1462. Springer, Heidelberg, Aug. 1998, pp. 1–12.

[5] "MEGA Security White Paper," https://mega.nz/SecurityWhitepaper.pdf, January 2020, visited on October 22, 2021.

[6] M. Haller, "MEGA: From bad practice to loss of confidentiality and integrity," 2022.

[7] A. P. K. Dalskov and C. Orlandi, "Can you trust your encrypted cloud?: An assessment of SpiderOakONE's security," in *ASIACCS 18*, J. Kim, G.-J. Ahn, S. Kim, Y. Kim, J. López, and T. Kim, Eds. ACM Press, Apr. 2018, pp. 343–355.

[8] K. K. Niehage, "Cryptographic vulnerabilities and other shortcomings of the nextcloud server side encryption as implemented by the default encryption module," Cryptology ePrint Archive, Report 2020/1439, 2020, https://eprint.iacr.org/2020/1439.

[9] V. Klima and T. Rosa, "Attack on private signature keys of the OpenPGP format, PGP(TM) programs and other applications compatible with OpenPGP," Cryptology ePrint Archive, Report 2002/076, 2002, https://eprint.iacr.org/2002/076.

[10] L. Bruseghini, K. G. Paterson, and D. Huigens, "Victory by KO: Attacking OpenPGP using key overwriting," in *ACM CCS 2022*. ACM Press, 2022.

[11] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults (extended abstract)," in *EUROCRYPT'97*, ser. LNCS, W. Fumy, Ed., vol. 1233. Springer, Heidelberg, May 1997, pp. 37–51.

[12] A. K. Lenstra, "Memo on RSA signature generation in the presence of faults," 1996, available from the author: arjen.lenstra@epfl.ch.

[13] A. Albertini, T. Duong, S. Gueron, S. Kölbl, A. Luykx, and S. Schmieg, "How to abuse and fix authenticated encryption without key commitment," Cryptology ePrint Archive, Report 2020/1456, 2020, https://eprint.iacr.org/2020/1456.

[14] Y. Dodis, P. Grubbs, T. Ristenpart, and J. Woodage, "Fast message franking: From invisible salamanders to encryptment," in *CRYPTO 2018, Part I*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10991. Springer, Heidelberg, Aug. 2018, pp. 155–186.

[15] J. Len, P. Grubbs, and T. Ristenpart, "Partitioning oracle attacks," in *USENIX Security 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, Aug. 2021, pp. 195–212.

[16] H. Böck, J. Somorovsky, and C. Young, "Return of bleichenbacher's oracle threat (ROBOT)," in *USENIX Security 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, Aug. 2018, pp. 817–849.

[17] T. Jager, S. Schinzel, and J. Somorovsky, "Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption," in *ESORICS 2012*, ser. LNCS, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459.   Springer, Heidelberg, Sep. 2012, pp. 752–769.

[18] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews, "Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks," in *USENIX Security 2014*, K. Fu and J. Jung, Eds.   USENIX Association, Aug. 2014, pp. 733–748.

[19] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom, "The 9 lives of bleichenbacher's CAT: New cache ATtacks on TLS implementations," in *2019 IEEE Symposium on Security and Privacy*.   IEEE Computer Society Press, May 2019, pp. 435–452.

[20] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *ACM CCS 2014*, G.-J. Ahn, M. Yung, and N. Li, Eds.   ACM Press, Nov. 2014, pp. 990–1003.

[21] M. Borgmann, T. Hahn, M. Herfert, T. Kunz, M. Richter, U. Viebeg, and S. Vowe, "On the security of cloud storage services," Fraunhofer SIT, Tech. Rep., 2012.

[22] S. Messmer, J. Rill, D. Achenbach, and J. Müller-Quade, "A novel cryptographic framework for cloud file systems and cryfs, a provably-secure construction," in *Data and Applications Security and Privacy XXXI*, G. Livraga and S. Zhu, Eds.   Cham: Springer International Publishing, 2017, pp. 409–429.

[23] C. Boyd, G. T. Davies, K. Gjøsteen, H. Raddum, and M. Toorani, "Security notions for cloud storage and deduplication," in *ProvSec 2018*, ser. LNCS, J. Baek, W. Susilo, and J. Kim, Eds., vol. 11192.   Springer, Heidelberg, Oct. 2018, pp. 347–365.

[24] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *FC 2010 Workshops*, ser. LNCS, R. Sion, R. Curtmola, S. Dietrich, A. Kiayias, J. M. Miret, K. Sako, and F. Sebé, Eds., vol. 6054.   Springer, Heidelberg, Jan. 2010, pp. 136–149.

[25] W. Chen and R. A. Popa, "Metal: A metadata-hiding file-sharing system," in *NDSS 2020*.   The Internet Society, Feb. 2020.

[26] W. Chen, T. Hoang, J. Guajardo, and A. A. Yavuz, "Titanium: A metadata-hiding file-sharing system with malicious security," Cryptology ePrint Archive, Report 2022/051, 2022, https://eprint.iacr.org/2022/051.

[27] D. Whiting, R. Housley, and N. Ferguson, "Counter with CBC-MAC (CCM)," RFC 3610, Sep. 2003. [Online]. Available: https://www.rfc-editor.org/info/rfc3610

[28] S. Frankel, K. R. Glenn, and S. G. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec," RFC 3602, Sep. 2003. [Online]. Available: https://www.rfc-editor.org/info/rfc3602

[29] J. Jonsson, "On the security of CTR + CBC-MAC," in *SAC 2002*, ser. LNCS, K. Nyberg and H. M. Heys, Eds., vol. 2595.   Springer, Heidelberg, Aug. 2003, pp. 76–93.

[30] H. L. Garner, "The Residue Number System," in *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, ser. IRE-AIEE-ACM '59 (Western).   New York, NY, USA: Association for Computing Machinery, 1959, p. 146–153. [Online]. Available: https://doi.org/10.1145/1457838.1457864

[31] G. D. Micheli and N. Heninger, "Recovering cryptographic keys from partial information, by example," Cryptology ePrint Archive, Report 2020/1506, 2020, https://eprint.iacr.org/2020/1506.

[32] N. A. Howgrave-Graham, "Computational Mathematics Inspired by RSA," Ph.D. dissertation, 1998.

[33] A. May, "Using LLL-reduction for solving RSA and factorization problems," ser. ISC, P. Q. Nguyen and B. Vallée, Eds.   Springer, Heidelberg, 2010, pp. 315–348.

[34] A. Joux, "Authentication Failures in NIST version of GCM," p. 3, 2006, https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/800-38-series-drafts/gcm/joux_comments.pdf.

[35] B. Huang, "Cache-collision timing attacks against AES-GCM," 2010, master's thesis. http://udspace.udel.edu/handle/19716/9765.

[36] E. Kasper and P. Schwabe, "Faster and timing-attack resistant AES-GCM," Cryptology ePrint Archive, Report 2009/129, 2009, https://eprint.iacr.org/2009/129.

[37] N. Ferguson, "Authentication weaknesses in GCM. Comments submitted to NIST Modes of Operation Process," 2005, http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/CWC-GCM/Ferguson2.pdf.

[38] S. Gueron and V. Krasnov, "The fragility of AES-GCM authentication algorithm," Cryptology ePrint Archive, Report 2013/157, 2013, https://eprint.iacr.org/2013/157.

[39] P. Rogaway and T. Shrimpton, "A provable-security treatment of the key-wrap problem," in *EUROCRYPT 2006*, ser. LNCS, S. Vaudenay, Ed., vol. 4004.   Springer, Heidelberg, May / Jun. 2006, pp. 373–390.

[40] M. Bellare and P. Rogaway, "Optimal asymmetric encryption," in *EUROCRYPT'94*, ser. LNCS, A. D. Santis, Ed., vol. 950.   Springer, Heidelberg, May 1995, pp. 92–111.

[41] M. J. Dworkin *et al.*, "NIST SP 800-38B. Recommendation for block cipher modes of operation: The CMAC mode for authentication," 2016.

[42] M. Dürmuth, T. Güneysu, M. Kasper, C. Paar, T. Yalcin, and R. Zimmermann, "Evaluation of standardized password-based key derivation against parallel processing platforms," in *ESORICS 2012*, ser. LNCS, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459.   Springer, Heidelberg, Sep. 2012, pp. 716–733.

[43] S. Jarecki, H. Krawczyk, and J. Xu, "OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks," in *EUROCRYPT 2018, Part III*, ser. LNCS, J. B. Nielsen and V. Rijmen, Eds., vol. 10822.   Springer, Heidelberg, Apr. / May 2018, pp. 456–486.

[44] CFRG, "Results of the PAKE selection process," https://datatracker.ietf.org/meeting/interim-2020-cfrg-01/materials/slides-interim-2020-cfrg-01-sessa-results-of-the-pake-selection-process-00, April 2020.

[45] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring Polynomials with Rational Coefficients," *Mathematische Annalen*, vol. 261, no. ARTICLE, pp. 515–534, 1982.

[46] P. Q. Nguyen and D. Stehlé, "LLL on the Average," in *Algorithmic Number Theory*.   Springer Berlin Heidelberg, 2006, pp. 238–256.

APPENDIX A
LATTICE CRYPTANALYSIS FOR RSA KEY RECOVERY ATTACK

We briefly describe the straightforward application of a low-dimensional lattice attack adapted from Section 4.2.2 of Gabrielle and Heninger [31]. The attack can be used to recover up to 341 of the unknown least significant bits of the RSA prime factor q after the most significant bits have been determined using the key recovery attack in Section III.

Let $\hat{q}_2$ be the leftmost $(1024 - l)$ bits of q and q_1 the remaining l unknown bits. Let $q_2 \leftarrow \hat{q}_2 \cdot 2^l$ such that $q = q_2 + q_1$. On a high level, we rewrite the problem of recovering q_1 to the task of finding small roots of a polynomial. For this purpose, we consider the following three polynomials f_1, f_2, and f_3 over $\mathbb{Z}$, which all have the small root q_1 modulo q:

$$
\begin{aligned}
f_1(x) &= x \cdot (q_2 + x), & f_1(q_1) &= q_1 \cdot (q_2 + q_1) \equiv_q 0 \\
f_2(x) &= q_2 + x, & f_2(q_1) &= q_2 + q_1 \equiv_q 0 \\
f_3(x) &= N, & f_3(q_1) &= N \equiv_q 0
\end{aligned}
$$

We observe that every linear combination of these polynomials has the same root q_1 modulo q. We use this observation to construct the following lattice basis B, where we put the coefficient vectors of the previous polynomials in the rows and scale the first column by L^2 and the second by L for $L = 2^l$:

$$
B = \begin{bmatrix} L^2 & Lq_2 & 0 \\ 0 & L & q_2 \\ 0 & 0 & N \end{bmatrix}
$$

The column scaling ensures that the L1 norm of any vector in the lattice is an upper bound on the value of the unscaled polynomial corresponding to the coefficient vector when evaluated at q_1. Consequently, if we can find a vector w with $||w||_1 < q$ in the lattice, then there is a corresponding unscaled polynomial g, such that $g(q_1) < q$. Since q_1 is a root of g

modulo q by construction, it follows that $g(q_1) = 0$ over the integers.

We can efficiently recover the missing bits q_1 of q by factoring the polynomial. The LLL algorithm [45] finds an exponential approximation of the shortest vector in polynomial time. Nguyen and Stehlé showed in [46] that the vector w found by LLL satisfies $||w||_2 \leq 1.02^n \det(B)^{1/n}$ on average for random lattices. For our lattice basis B with dimension $n = 3$ and determinate $\det B = L^3 \cdot N$, we derive as the condition for $||w||_1 < q$ that $l \leq \log_2(N^{1/6})$. We can therefore recover up to $l = 341$ bits for RSA-2048 using this simple lattice. Higher-dimensional lattices would allow us to decrease the number of queries to 512 as shown in [32], [33].

APPENDIX B
GUESS-AND-PURGE BLEICHENBACHER ATTACK: DETAILED DESCRIPTION

We first specify MEGA's custom padding and the oracle that it exposes. Then, we extend Bleichenbacher's attack steps for PKCS#1 v1.5 padding [4] to work on MEGA's custom padding despite an unknown prefix. We conclude by analyzing the correctness and complexity of our GaP-Bleichenbacher attack.

A. MEGA's Padding Scheme

When no long-term `Curve25519` chat key is available, MEGA uses RSA encryption as a fallback method to exchange one or more 16-byte chat keys, concatenated together to K. The encryption procedure applies the following padding:

$$\mathsf{MEGA\_PAD}(K) := t\|L\|K\|P,$$

where t is a two-byte prefix, L encodes the byte length of the chat key(s) K in two bytes (with big-endian encoding), and P is random padding that extends the message to 256 bytes. The server uses $t \leftarrow 0^{16}$.

The client parses the above message after RSA-2048 decryption as follows. First, it removes and ignores the prefix t. Second, it recovers the key length and checks that it is a multiple of 16, the byte length of a single chat key. If this check succeeds, the client extracts the chat key(s) and discards the padding. Otherwise, it throws an exception and reports the error to the server.

The padding removal is successful iff L is of the form $0^8\|\{0,1\}^4\|0^4$. Our GaP-Bleichenbacher attack mainly uses the zero prefix. The rightmost four zero bits of L can potentially be used for further optimizations.

B. Attack Description

This section explains our extension of the original attack steps from [4] to account for MEGA's leakage pattern and the unknown prefix.

To stay close to the notation of [4], let $c = m^e \bmod N$ be the RSA ciphertext of a target message m. Let $B \leftarrow 256^{252}$ be the power of two that exceeds the largest possible unencoded plaintext by one. We call a message *conforming* when it is correctly padded. Let m_0 be the conforming multiple of the
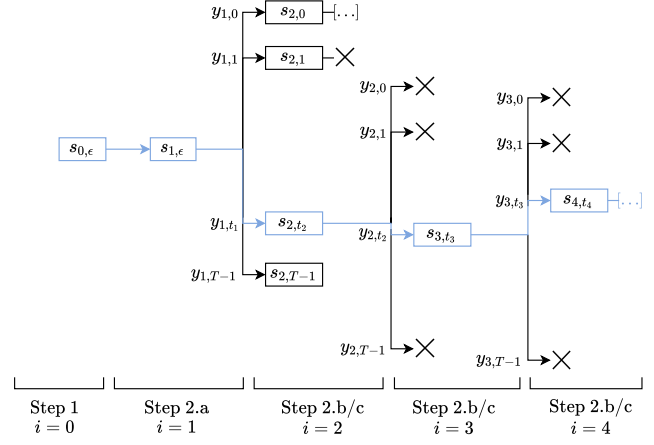

Fig. 9. Visualization of the Guess-and-Purge strategy for our modified Bleichenbacher attack with $T = 2^{16}$ prefix guesses $t \in \{0,1\}^{16}$. The blue path up to iteration i corresponds to a workload $w = (\mathcal{M}_{i,t_i}, \mathcal{H}_{i,t_i})$, where $\mathcal{M}_{i,t_i}$ is the solution interval implicitly associated to the path that depends on the previous choice of multipliers $(s_{0,\epsilon}, s_{1,\epsilon}, s_{2,t_2}, \dots, s_{i,t_i})$ stored in $\mathcal{H}_{i,t_i}$. Paths with empty intervals (marked by a cross) are not extended.

target plaintext. If m_0 is of known byte length l_{m_0}, then m_0 has to be in the interval $[l_{m_0} \cdot B, (l_{m_0} + 1) \cdot B - 1]$ due to the padding scheme.

The execution of our attack can be visualized as a tree (see Figure 9), where nodes represent a multiplier and the corresponding prefix guess. Every node has an associated set of intervals of candidate decryptions of our target ciphertext c. If the prefix guesses on the path to this node were correct, then there is some interval that contains the decryption m of our target ciphertext. In every iteration, we select a new multiplier for every leaf node. Each multiplier has $T = 2^{16}$ possible prefixes. We add new successor nodes if the interval of possible plaintext decryptions is still non-empty. Otherwise, we know that there was a wrong prefix guess on this path, and we no longer extend this path in the next iteration (marked with a cross in Figure 9). The multipliers selected for different leaves may differ since they depend on the previous multipliers and prefixes.

We introduce the concept of *workloads* for our variant of Bleichenbacher's attack. A workload corresponds to a path in the attack tree. In other words, it stores the state of one possible solution path, including all prefix guesses that led to the current state and a set of intervals. We formalize this as follows. Let $\mathcal{H}_{i,t_i} = (s_{0,t_0}, s_{1,t_1}, \dots, s_{i,t_i})$ denote the history of multipliers on the path. The guesses $t_j \in \{0,1\}^{16}$ for all $j \in [0, i]$ are the leftmost two bytes of $s_{j,t_j} \cdot m_0 \bmod N$ after left padding the result with zero bytes to 256 B. Let $\mathcal{M}_{i,t_i}$ denote the set of closed intervals after iteration i, resulting from the choice of multipliers. We define a workload w to be the tuple $(\mathcal{M}_{i,t_i}, \mathcal{H}_{i,t_i})$. Furthermore, we denote by $t_0 = t_1 = \epsilon$ that there is no prefix guess for the first two multipliers.

As explained in detail below, the multipliers are chosen independently of any prefix guess: *Step 1* chooses s_{0,t_0} randomly and *Step 2.a* linearly searches for s_{1,t_1}, starting from a value

derived from the initial bounds. For $k \in [2, i]$, *Step 2.c* chooses the multiplier $s_{k,t_k} \in \mathbb{Z}$ based on the shifted prefix guess $y_k \leftarrow 256^2 \cdot B \cdot t_k$ of the conforming message $s_{k,t_k} \cdot m_0 \bmod N$ and the previous intervals $\mathcal{M}_{k-1,t_{k-1}}$ such that s_{k,t_k} reduces the size of the possible solution intervals adequately. We guess the shifted prefix y_k before selecting s_{k,t_k}.

We remark that our indices for multipliers and prefix guesses are only unique within the same workload. We do not use globally unique identifiers to avoid a cluttered notation. In particular, the multipliers and prefixes for different workloads in the same iteration do not have to be equal.

Finally, we introduce the oracle $\mathcal{O}_{c_0}(s)$ which returns true iff the RSA chat key decryption succeeds for the ciphertext c_0 multiplied by $s^e \bmod N$.

For our GaP-Bleichenbacher attack, we perform *Step 1* once at the beginning of the attack. For every iteration i, we perform *Step 2* to *Step 4* for every workload $w = (\mathcal{M}_{i-1,t_{i-1}}, \mathcal{H}_{i-1,t_{i-1}}) \in \mathcal{W}_{i-1}$.

Step 1: Blinding. Given a target ciphertext $c = m^e \bmod N$, we sample random multipliers $s_{0,\epsilon}$ until $\mathcal{O}_{c_0}(s_{0,\epsilon})$ returns true. For the first successful value $s_{0,\epsilon}$, we set:

$$c_0 \leftarrow (c \cdot (s_{0,\epsilon})^e) \bmod N$$
$$\mathcal{M}_{0,\epsilon} \leftarrow \{[pt_{min}, pt_{max} - 1]\}$$
$$\mathcal{H}_{0,\epsilon} \leftarrow (s_{0,\epsilon})$$
$$\mathcal{W}_0 \leftarrow \{(\mathcal{M}_{0,\epsilon}, \mathcal{H}_{0,\epsilon})\}$$
$$i \leftarrow 1$$

where $pt_{min} \leftarrow 0$ and $pt_{max} \leftarrow 241 \cdot B$ are the smallest resp. largest possible plaintexts (including length encoding) which conform to MEGA's padding. The subsequent attack recovers $m_0 \leftarrow (s_{0,\epsilon} \cdot m) \bmod N$, which is the decryption of c_0. We do not need any prefix guess (as indicated by ϵ) as $\mathcal{M}_{0,\epsilon}$ contains a single interval specifying the maximum range of conforming plaintexts.

Step 2: Searching for a multiplier satisfying $\mathcal{O}_{c_0}$.

Step 2.a: Starting the search. For $i = 1$, we have only a single workload with $\mathcal{M}_{0,\epsilon} = \{[a, b]\}$ (in the generic case, $a = 0$ and $b = 241 \cdot B - 1$). We search for the smallest $s_{1,\epsilon} \geq N/(b+1)$ such that $\mathcal{O}_{c_0}(s_{1,\epsilon})$ returns true.

Step 2.b: Sequential searching with $|\mathcal{M}_{i-1,t_{i-1}}| > 1$. For $i > 1$ and more than one interval left, where $s_{i-1,t_{i-1}} \in \mathcal{H}_{i-1,t_{i-1}}$, we search for the smallest $s_{i,t_i} > s_{i-1,t_{i-1}}$ where $\mathcal{O}_{c_0}(s_{i,t_i})$ returns true.

Step 2.c: Interval-based searching with $|\mathcal{M}_{i-1,t_{i-1}}| = 1$. For $i > 1$ and exactly one interval $[a, b]$ left, where $s_{i-1,t_{i-1}} \in \mathcal{H}_{i-1,t_{i-1}}$, we iterate over all possible prefix guesses $t_i \in \{0, 1\}^{16}$ with the corresponding shifted values $y_i \leftarrow 256^2 \cdot B \cdot t_i$ of the still unknown value $s_{i,t_i} \cdot m_0 \bmod N$. For every prefix guess, we search the smallest pair of variables r_i and s_{i,t_i} which satisfy the following

two constraints as well as $\mathcal{O}_{c_0}(s_{i,t_i})$. Due to the choice of r_i and s_{i,t_i}, we approximately halve the interval $[a, b]$ in *Step 3*.

We start incrementing r_i from

$$r_i \geq \frac{2 \cdot b \cdot s_{i-1,t_{i-1}} - pt_{min} - y_i}{N}.$$

For every r_i value, we try the following multipliers:

$$\frac{pt_{min} + r_i \cdot N + y_i}{b} \leq s_{i,t_i} < \frac{pt_{max} + r_i \cdot N + y_i}{a}.$$

Section B-C discusses the reasoning for this search procedure in detail. However, the intuition is as follows: there exists at least one solution because we are guaranteed to find a conforming s_{i,t_i} value for the workload with all correct prefix guesses since our procedure then performs *Step 2.c* from the original Bleichenbacher attack without an unknown prefix. If we end up using another multiplier s_{i,t_i} for a wrong prefix guess, this still reduces our intervals. Although this s_{i,t_i} might not eliminate as many plaintext candidates as the one for the correct prefix, it is still a correct multiplier because the oracle decision is independent of our prefix guess.

Step 3: Narrowing the set of solutions. For all intervals $[a, b] \in \mathcal{M}_{i-1,t_{i-1}}$ and the multiplier guess history $\mathcal{H}_{i-1,t_{i-1}} = (s_{0,t_0}, s_{1,t_1}, \ldots, s_{i-1,t_{i-1}})$, we update the bounds for every prefix guess $t^* \in \{0, 1\}^{16}$ of $(s_{i,t_i} \cdot m_0) \bmod N$ and the corresponding $y^* \leftarrow 256^2 \cdot Bt^*$. We update the intervals and prefix guess history as follows, where $s_{i,t^*} \leftarrow s_{i,t_i}$:

$$\mathcal{M}_{i,t^*} \leftarrow \cup_{a,b,r} \{[a', b']\}$$
$$\mathcal{H}_{i,t^*} \leftarrow (s_{0,t_0}, s_{1,t_1}, \ldots, s_{i-1,t_{i-1}}, s_{i,t^*}).$$

In the above equations, the bounds a' and b' are specified as follows:

$$a' \leftarrow \max\left(a, \left\lceil \frac{pt_{min} + r \cdot N + y^*}{s_{i,t_i}} \right\rceil\right)$$
$$b' \leftarrow \min\left(b, \left\lfloor \frac{pt_{max} - 1 + r \cdot N + y^*}{s_{i,t_i}} \right\rfloor\right).$$

for all r values in the following range:

$$\frac{a \cdot s_{i,t_i} - pt_{max} + 1 - y^*}{N} \leq r \leq \frac{b \cdot s_{i,t_i} - pt_{min} - y^*}{N}$$

We add a new workload $(\mathcal{M}_{i,t^*}, \mathcal{H}_{i,t^*})$ to $\mathcal{W}_i$ whenever $\mathcal{M}_{i,t^*} \neq \emptyset$.

It is necessary to consider all possible prefixes $t^* \in \{0, 1\}^{16}$ to guarantee the existence of a fully correct prefix guess history $t_0, t_1, \ldots, t^*$ (which is implicitly stored in $\mathcal{H}_{i,t^*}$). For instance, if we would only use the prefix t_i for which *Step 2.c* found the multiplier s_{i,t_i}, then the target plaintext m_0 might not be in any of the remaining intervals because the first

two bytes t^* of $s_{i,t_i} \cdot m_0 \bmod N$ are not equal to t_i.

Step 4: Computing the solution. If there is only one workload $\mathcal{W}_{i-1} = \{(\mathcal{M}_{i-1,t_{i-1}}, \mathcal{H}_{i-1,t_{i-1}})\}$ and only one interval $\mathcal{M}_{i-1,t_{i-1}} = \{[a,a]\}$ containing a single value, then we have $a = m_0$ and return the solution $m \leftarrow a \cdot (s_{0,\epsilon})^{-1} \bmod N$.

Otherwise, we continue executing the attack. If we did not yet iterate over all workloads in $\mathcal{W}_{i-1}$, we go to *Step 2* with the next workload from that set. If there is no workload left for iteration i, then we set $i \leftarrow i + 1$ and continue with *Step 2* for the new set of workloads $\mathcal{W}_i$.
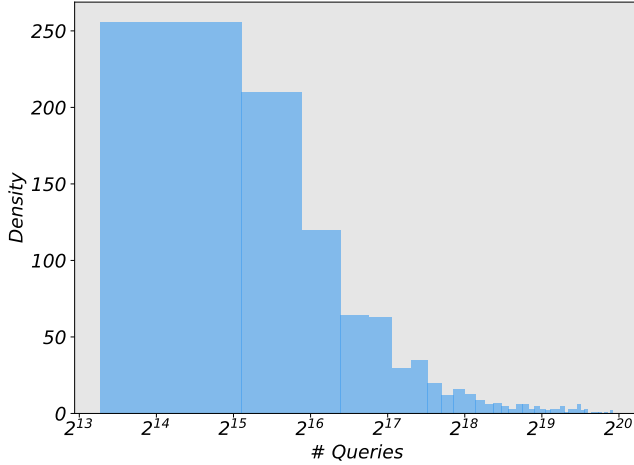

Fig. 10. Density plot of the number of oracle queries.

C. Correctness

The extension of Bleichenbacher's attack [4] to MEGA's padding scheme is challenging because of the unknown prefix y_i. We adapt the equations from [4] to account for y_i as follows:

$$\mathcal{O}_{c_0}(s_i) \implies \exists r \in \mathbb{Z}, \exists t^* \in \{0,1\}^{16} \text{ such that}$$
$$pt_{min} \le s_i \cdot m_0 - rN - 256^2 \cdot Bt^* \le pt_{max} - 1. \quad (1)$$

Let $r \in \mathbb{Z}$ and $t^* \in \{0,1\}^{16}$ with the corresponding shifted value $y^* \leftarrow 256^2 \cdot Bt^*$ be values satisfying the right-hand side of the implication in Equation 1, for some s_i such that $\mathcal{O}_{c_0}(s_i)$. We solve the inequalities for m_0 to derive the bounds for m_0 used in *Step 3* to narrow down the intervals:

$$\frac{pt_{min} + r \cdot N + y^*}{s_i} \le m_0 \le \frac{pt_{max} - 1 + r \cdot N + y^*}{s_i}. \quad (2)$$

Furthermore, we can derive the bounds for r used in *Step 3* from Equation 1 by using $a \le m_0 \le b$ for some interval $[a,b] \in \mathcal{M}_{i,t}$ and $m_0 \in [a,b]$:

$$\frac{s_i \cdot a - pt_{max} + 1 - y^*}{N} \le \frac{s_i \cdot m_0 - pt_{max} + 1 - y^*}{N} \le r$$
$$r \le \frac{s_i \cdot m_0 - pt_{min} - y^*}{N} \le \frac{s_i \cdot b - pt_{min} - y^*}{N}. \quad (3)$$

The bounds for the multipliers used in *Step 2* can be derived similarly.

We can use the above statements to prove the correctness of our algorithm by induction over i. Let

$$T(i) \equiv (\exists t_i \in \{0,1\}^{16})(\exists [a,b] \in \mathcal{M}_{i,t_i}) \text{ s.t. } m_0 \in [a,b]$$

be our induction hypothesis stating that in every iteration, there exists at least one interval containing the target message m_0. Since the last interval has length one, this implies that we find the correct plaintext m_0 and, thus, return the decryption m of the target ciphertext c.

The base case $T(0)$ trivially holds because $\mathcal{M}_{0,\epsilon} = \{[pt_{min}, pt_{max}]\}$ and $m_0 \in [pt_{min}, pt_{max}]$ by definition since pt_{min} and pt_{max} are the smallest, respectively largest, plaintext values.

We assume $T(i - 1)$ for the induction step and show $T(i)$. *Step 2* uses some s_i with $\mathcal{O}_{c_0}(s_i)$ by construction. Therefore, by Equation 1 there exist r and $t^* \in \{0,1\}^{16}$ such that the right-hand side of the implication holds. By Equation 3, we know that the range of r values used in *Step 3* contains the correct one. Furthermore, we iterate over all $t^* \in \{0,1\}^{16}$ and add intervals to $\mathcal{M}_{i,t^*}$. Therefore, for the correct r and t^*, we narrow $[a,b]$ to $[a',b']$ in *Step 3* where the bounds from Equation 2 guarantee that $m_0 \in [a',b']$. We conclude the induction proof by noting that $[a',b'] \in \mathcal{M}_{i,t^*}$ implies $T(i)$.

D. Complexity

The density histogram in Figure 10 shows that our GaP-Bleichenbacher has a query complexity of $\mu \approx 2^{16.9}$ on average with a comparatively high standard deviation of $\sigma \approx 2^{17.3}$. A quarter of all runs only require 2^{14} queries, but the distribution has a long tail, and we aborted 71 out of 1000 runs because they exceeded our cutoff of 10^6 queries. We use the Freedman-Diaconis binning rule to decide on an appropriate number of bins of equal width.

The query complexity of our GaP-Bleichenbacher attack is significantly lower than executing 2^{16} classic Bleichenbacher attacks for every prefix guess. Figure 11 visualizes the core reason: every conforming multiplier s_{i,t_i} allows us to detect workloads with an invalid prefix guess in $\mathcal{H}_{i,t_i}$ because they result in an empty solution interval $\mathcal{M}_{i,t_i} = \emptyset$. The stacked bar plot shows that the first multiplier $s_{1,\epsilon}$ adds approximately 2500 plausible prefix guesses. The next multiplier s_{2,t_2} eliminates more than 95% of the wrong guesses shown with a blue bar in Figure 10; the remaining workloads are gray with an error bar showing the standard deviation. As the solution intervals narrow, every multiplier adds fewer new workloads while following multipliers eliminate wrong guesses quickly. We do not require more queries than the classic Bleichenbacher attack after approximately 28 multipliers because only a single workload remains.

Our optimizations are another reason for this good performance compared to the original attack. We use dynamic programming to avoid repeated queries for different workloads. Furthermore, we utilize that MEGA's padding ends in
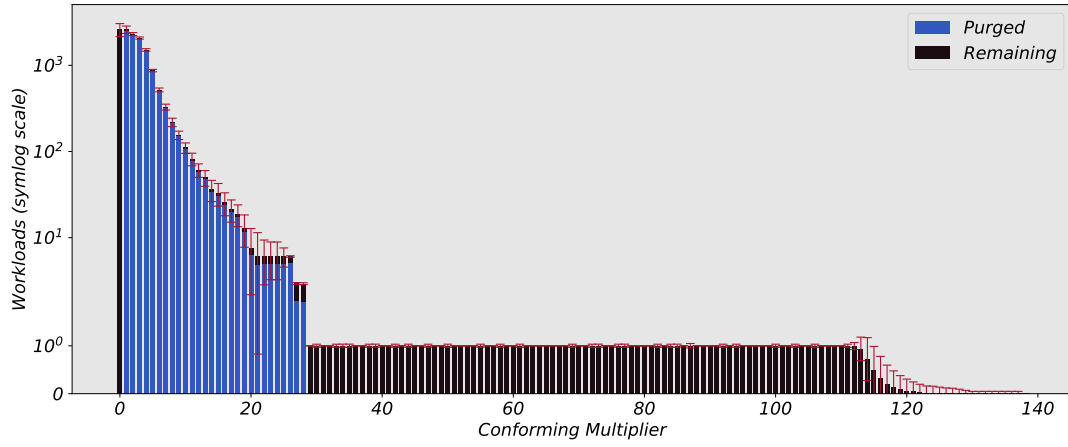
Fig. 11. Number of purged and remaining workloads for every multiplier.

random bytes, which we do not need to recover. Therefore, we terminate the attack as soon as the interval of possible plaintexts has a stable prefix that includes all message bits.

E. Conclusion

The GaP-Bleichenbacher attack extends the original attack on PKCS#1 v1.5 padding to MEGA's custom padding with unknown prefixes. We evaluated this variant to require $2^{16.9}$ queries on average despite guessing a two-byte prefix. The attack is still challenging to exploit in practice because it requires a substantial number of queries. The adversary is also challenging to instantiate in practice. Despite the theoretical nature of the GaP-Bleichenbacher attack, it still points out two weaknesses of MEGA's system. First, implementing custom padding schemes instead of using provably secure standards is dangerous. Second, key reuse allows the adversary to decrypt arbitrary RSA ciphertexts using legacy code.